

Scaling Functional Synthesis and Repair

Thèse N° 8716

Présentée le 11 janvier 2019

à la Faculté informatique et communications

Laboratoire d'analyse et de raisonnement automatisés

Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

EMMANOUIL KOUKOUTOS

Acceptée sur proposition du jury

Prof. C. Koch, président du jury

Prof. V. Kuncak, directeur de thèse

Prof. S. Khurshid, rapporteur

Prof. D. Marinov, rapporteur

Prof. M. Odersky, rapporteur

2019

Acknowledgements

First and foremost, I would like to thank my advisor Viktor Kuncak, for accepting me into his lab, and generously providing his guidance, knowledge, encouragement and support during my PhD studies. This thesis would not have been possible without him.

Second, I would like to thank my thesis committee members Christoph Koch, Martin Odersky, Darko Marinov and Sarfraz Khurshid, for taking the time to evaluate my thesis, and for their valuable and extensive comments on it, which helped it improve greatly to its current form.

Third, I would like to thank my colleagues Etienne Kneuss, Mukund Raghothaman, Ravishadran Kandhadai Madhavan, Régis Blanc, Mikael Mayer, Nicolas Voirol, Romain Edelmann, Eva Darulova, Andreas Pavlogiannis, Georg Schmid, Jad Hamsa, Romain Ruetschi, and Sarah Sallinger, for our collaboration, exchange of ideas, and the welcoming and creative atmosphere they maintained in the lab. A special thanks goes to Etienne for our great collaboration, and his guidance and patience when he was helping me acclimate during my first months in the lab. Also, I would like to thank Fabien Salvi, Yvette Gallay and Sylvie Jankow, for helping all of us to work unobstructed with their technical and administrative support.

Next, I would like to thank all my friends in Lausanne for their companionship and support, and my friends away from here for their continued love and support, despite how little time I admittedly devoted to them during times.

Finally, a big thanks to my extended family for their unconditional love and support. I would like to especially thank my father for teaching me that the world can be understood with logic and inspiring me to pursue scientific studies as well as this PhD. Lastly, I would like to thank my mother, who left us too young and too long ago, for making me the person I am today. I know she would have been much prouder of this achievement even than I am myself.

Lausanne, 25 October 2018

M. K.

Abstract

Program synthesis was first proposed a few decades ago, but in the last decade it has gained increased momentum in the research community. The increasing complexity of software has dictated the urgent need for improved supporting tools that verify the software’s correctness, and that automatically generate code from a formal contract provided by the programmer, along with a proof of the generated code’s correctness. In addition, recent technological developments have provided tools that have enabled researchers to revisit the synthesis problem. The recent rise of SMT solvers has given synthesis tools a reliable and automated way to verify synthesized programs against contracts. The introduction of counter-example guided inductive synthesis has provided researchers with a flexible synthesis algorithm that they can adapt according to their specific domain.

In this dissertation, we develop new algorithms to synthesize recursive functional programs with algebraic data types from formal specifications and/or input-output examples. We manage to scale beyond the reach of other similar tools to synthesize nontrivial functional programs, with a focus on data structure transformations.

First, we address the problem of precisely specifying the desired space of candidate programs, described by context free grammars (CFGs). We implement and evaluate a method for reducing the program space by describing axioms of the target language and other domain-specific restrictions on the level of the CFG, without explicitly generating and rejecting undesirable programs. We provide a method that extracts a program model from a corpus of code and that builds a probabilistic CFG from it. We showcase the usefulness, both individually and in tandem, of these methods.

Second, we develop an algorithm to efficiently traverse a possibly unbounded space of candidate programs generated from a probabilistic CFG. This algorithm is an implementation of the A* best-first search algorithm on the derivation graph generated from the CFG, with a number of domain-specific optimizations. We evaluate the efficiency of the algorithm as well as the effectiveness of the optimizations.

Finally, we describe a program repair framework that locates and fixes bugs in erroneous functional programs. Our novel fault localization technique detects erroneous snippets with concrete execution and eliminates false positives by analyzing dependencies between execution traces. After the erroneous code snippet is discovered, a modified version of our synthesis algorithm generates fixes for it by introducing modifications to the original erroneous code.

Keywords: functional programming, program synthesis, program repair, formal grammars

Résumé

La synthèse de programmes a été proposée il y a déjà quelques décennies, mais elle a récemment gagné en popularité dans le milieu de la recherche. La complexité toujours croissante des logiciels impose un besoin urgent pour de meilleurs outils de développement qui vérifient la validité de ceux-ci, et qui génèrent automatiquement une implémentation à partir de contrats formels fournis par le développeur, ainsi qu'une preuve de la validité du logiciel généré. De plus, de récents développements techniques ont permis aux chercheurs de revisiter le problème de la synthèse de programmes. L'essor des solveurs SMT a fourni aux outils de synthèse un moyen robuste et automatique de vérifier les programmes synthétisés contre leur spécification. L'apparition de la synthèse inductive guidée par contre-exemples a fourni aux chercheurs un algorithme de synthèse flexible qui peut être adapté à chaque domaine spécifique.

Dans cette dissertation, nous développons de nouveaux algorithmes pour synthétiser des programmes fonctionnels récursifs sur des types algébriques de données à partir de spécifications formelles et/ou d'exemples d'entrées-sorties. Nos algorithmes surpassent la portée d'outils similaires et parviennent à synthétiser des programmes non triviaux avec un accent sur les transformations de structures de données.

Tout d'abord, nous abordons le problème de la spécification précise de l'espace de recherche de programmes candidats, que nous décrivons à l'aide de grammaires non contextuelles (CFGs). Nous implémentons et évaluons une technique de réduction de l'espace de recherche en décrivant certains axiomes du langage ciblé ainsi que d'autres restrictions spécifiques au domaine particulier directement au niveau de la grammaire non contextuelle. Ceci nous évite d'avoir à explicitement construire puis rejeter des programmes indésirables. Nous fournissons une méthode qui extrait un modèle de programme d'un corpus de code, et qui génère une grammaire non contextuelle stochastique à partir de celui-ci. Nous démontrons l'utilité de ces deux techniques à la fois individuellement et en tandem.

Ensuite, nous développons un algorithme pour visiter efficacement un ensemble potentiellement infini de programmes générés à partir d'une grammaire non contextuelle stochastique. Cet algorithme est une implémentation de l'algorithme A* sur le graphe de dérivations généré à partir de la grammaire, augmenté de plusieurs optimisations spécifiques au domaine. Nous évaluons la performance de l'algorithme et l'efficacité des optimisations.

Finalement, nous décrivons un système de réparation de programmes qui localise et corrige les erreurs contenues dans des programmes fonctionnels erronés. Notre technique novatrice de localisation d'erreurs détecte les fragments erronés à l'aide d'exécutions concrètes puis

Acknowledgements

élimine les faux positifs en analysant les dépendances entre les différentes traces. Une fois le fragment coupable identifié, une version adaptée de notre algorithme de synthèse résout l'erreur en modifiant le code erroné originel.

Mots-clefs : programmation fonctionnelle, synthèse de programme, réparation de programme, grammaires formelles

Contents

Acknowledgements	iii
Abstract (English/Français)	v
List of figures	xiii
List of tables	xv
Introduction	1
1 Deductive Synthesis and the Leon Framework	7
1.1 Overview of Leon	7
1.2 Examples of Synthesis Problems	8
1.2.1 Distinct Number	8
1.2.2 Sorted List Insertion and List Sorting	9
1.2.3 Run-Length Encoding	11
1.2.4 Repairing Heap Merging	13
1.3 The Synthesis Problem	14
1.3.1 Path Conditions	15
1.4 Input Language	15
1.4.1 Syntax of PureScala	16
1.4.2 Synthesis Constructs	16
1.4.3 Symbolic Examples	18
1.4.4 Object-Oriented Extension	19
1.5 Deductive Synthesis	23
1.5.1 Rule Instantiation	24
1.5.2 Application of a Rule Instantiation	24
1.5.3 Solution Composition	25
1.5.4 Normalizing Rules	25
1.5.5 Synthesis Rules as Inference Rules	25
1.5.6 Definition of Synthesis Rules	26
1.5.7 Example of a Synthesis Graph	34
1.6 Conclusion	34
	ix

2	Term Grammars	37
2.1	Definition	37
2.2	Aspect Grammars	40
2.2.1	Example	41
2.2.2	Formalization	42
2.2.3	Definitions of Aspects	43
2.2.4	Comparison to other Grammar Formalisms	45
2.2.5	Implementation	47
2.3	Probabilistic Term Grammars	47
2.4	Generic Term Grammars	48
2.5	Built-In Grammar	49
2.5.1	Value Grammar	50
2.6	Custom Grammars	50
2.6.1	Grammar Files	50
2.7	Extracting Grammars from Corpus	53
2.7.1	Extractor for Plain Grammars	53
2.7.2	Extractor for Annotated Grammars	54
2.7.3	Extraction of Grammars from a Corpus of Leon Benchmarks	55
2.8	Extended Example	55
2.9	Conclusion	61
3	Term Exploration	63
3.1	Counter-Example Guided Inductive Synthesis	63
3.2	Symbolic Term Exploration	65
3.2.1	Sizes of Operators	67
3.3	Probabilistic Top-Down Term Enumeration	68
3.3.1	Derivation Trees	68
3.3.2	The Enumerate Algorithm	70
3.3.3	Nonterminal Horizons and A* Search	72
3.3.4	Computing the Horizon, $h(\bar{e})$	74
3.3.5	Optimizations	74
3.4	Evaluation	76
3.4.1	Aims	76
3.4.2	Description of Tables	77
3.4.3	Assessment of Results	78
3.5	Conclusion	81
4	Program Repair	85
4.1	Example	86
4.2	Overview of the Algorithm	87
4.3	Test Generation	88
4.3.1	Sources of Tests	88
4.3.2	Test Classification	88

4.3.3	Test Minimization	89
4.4	Fault Localization	90
4.4.1	Nondeterministic Evaluator	92
4.5	Similar Term Exploration	93
4.5.1	Similar Term Grammar	93
4.6	Verification	94
4.7	Characterization of Repairable Programs	94
4.8	Evaluation	95
4.9	Conclusion	100
5	Related Work	101
5.1	Synthesis Systems	101
5.2	Program Space Representations	104
5.3	Repair	105
6	Conclusions and Future Work	107
A	Synthesis Benchmarks by Specification	109
A.1	List.insert	109
A.2	List.delete	109
A.3	List.union	110
A.4	List.diff	110
A.5	List.split	111
A.6	List.listOfSize	111
A.7	SortedList.insert	111
A.8	SortedList.insertAlways	112
A.9	SortedList.delete	113
A.10	SortedList.union	113
A.11	SortedList.diff	114
A.12	SortedList.insertionSort	115
A.13	StrictSortedList.insert	115
A.14	StrictSortedList.delete	116
A.15	StrictSortedList.union	117
A.16	UnaryNumerals.add	117
A.17	UnaryNumerals.distinct	118
A.18	UnaryNumerals.mult	119
A.19	BatchedQueue.enqueue	119
A.20	BatchedQueue.dequeue	120
A.21	AddressBook.makeAddressBook	121
A.22	AddressBook.merge	122
A.23	RunLength.encode	123
A.24	Diffs.diffs	124

Contents

B	Synthesis Benchmarks by Example	127
B.1	UnaryNumerals.add	127
B.2	List.append	127
B.3	Calc.eval	128
B.4	Tree.countLeaves	129
B.5	Dictionary.replace	129
B.6	Dictionary.find	130
B.7	List.diffs	130
B.8	Expr.fv	131
B.9	UnaryNumerals.isEven	131
B.10	SortedList.insert	132
B.11	UnaryNumerals.mult	133
B.12	Tree.postorder	133
B.13	List.reserve	134
B.14	RunLength.encode	134
B.15	List.take	135
B.16	List.unzip	135
C	Repair Benchmarks	137
C.1	Compiler Benchmark	137
C.2	Heap Benchmark	142
C.3	List Benchmark	144
C.4	Numerical Benchmark	151
C.5	MergeSort Benchmark	152
	Bibliography	155
	Curriculum Vitae	163

List of Figures

1.1	Syntax of PureScala	17
1.2	Desugaring of synthesis holes	18
1.3	Definition of passes in Scala	19
1.4	Syntax modifications for object-oriented extensions	20
1.5	Desugaring of an object-oriented program	21
1.6	Generic synthesis rule as inference rule	26
1.7	Composition example of Section 1.5.3	26
1.8	Normalizing synthesis rules	27
1.9	Splitting synthesis rules	28
1.10	ADT splitting synthesis rules	29
1.11	Sketch for term exploration rule	29
1.12	Computing smaller arguments for recursive calls	29
1.13	Example of recursive call with two arguments changed	31
1.14	Synthesis graph for sorted list insertion	34
2.1	A simple expression language and its type system	39
2.2	Built-in grammar before processing	57
2.3	Part of grammar of figure 2.2 after processing	58
2.4	Plain custom grammar before preprocessing	59
2.5	Part of plain custom grammar after preprocessing	60
2.6	Part of annotated custom grammar after preprocessing	61
3.1	A tree of partial derivations for a PCFG	69
3.2	The operation of Enumerate(G, N) in Algorithm 3.5	70
4.1	Code and invocation graph for fv	89

List of Tables

3.1	Benchmarks for synthesis in automatic mode	79
3.2	Benchmarks for synthesis in manual mode	80
3.3	Benchmarks for synthesis by example in automatic mode	81
3.4	Demonstrating the effect of indistinguishability heuristic. Benchmarks are run with PTE in automatic mode	82
3.5	Demonstrating the effect of aspects. Benchmarks are run with STE in automatic mode	83
3.6	Demonstrating the effect of scores in the priority function. Benchmarks are run with PTE in manual mode	83
4.1	Benchmarks repaired by Leon: error types, sizes, and test generation times . .	98
4.2	Benchmarks repaired by Leon: localization and repair times	99

Introduction

Software development is the process of translating human intention into computer code. As humans and computers process completely different languages (if indeed the human has managed to express their intention with language), this process is tedious, error prone, and largely inaccessible to non-experts. Fortunately, the gap has been decreasing with the development of programming languages, as they are steadily becoming more high-level, i.e., closer to human thinking.

Despite this improvement, the core principle of programming remains the same: a qualified programmer has to manually translate their intentions (“what” needs to be done) into a computer algorithm (“how” it needs to be done). The ultimate goal in the field of *program synthesis* is to bridge this gap, by offering tools that translate the intention of the programmer into code executable by a machine. Intention is expressed by some form of specification that involves the input and desired output of the program. In these early stages of the field’s development, most systems accept specification that is itself expressed in a formal language, as opposed to a natural language. When a synthesis system generates a candidate solution, it tries to verify its correctness. The verification procedures vary across systems.

Although some core ideas for program synthesis were expressed already in the 1960s [Gre69, MW71], the research field has become much more active in roughly the last decade, following two major developments: the development and wide adoption of satisfiability-modulo-theory (SMT) solvers and the introduction of counter-example directed inductive synthesis (CEGIS). SMT solvers are automated theorem provers operating on specific logical theories. They attempt to prove or disprove that a logical formula is satisfiable by applying a set of decision procedures and heuristics; if the answer is positive, a satisfying assignment is produced as a certificate. SMT solvers have proven quite effective in solving many verification queries generated by synthesis, but more importantly, they are part of CEGIS. CEGIS is a synthesis algorithm that uses concrete inputs to accelerate the exploration of a space of candidate synthesis solutions, and to efficiently test those candidates for correctness. CEGIS is the heart of many synthesis systems and will be discussed in more detail in Chapter 3.

Our main goal in this dissertation is to synthesize recursive functional programs over unbounded datatypes. More specifically, we explore techniques that improve the efficiency of synthesis by reducing the set of solution candidates, as well as by accelerating the exploration

List of Tables

of this set. The language we handle is Turing-complete, which makes the synthesis problem in our context harder compared to other works focusing on less expressive fragments. In contrary, our choice of a functional language makes our programs easier to model and reason about due to the absence of mutable state and side-effects.

The specification for synthesis tasks we handle can be given as a logical predicate, a set of input-output examples, or a combination thereof.

Input-output examples map a finite subset of inputs to their corresponding outputs. Other inputs are free to be mapped to any output. For example, suppose we want to synthesize an algorithm which sorts a list of integers in ascending order. We could partially specify this algorithm with the following set of input-output examples:

List()	→	List()
List(1)	→	List(1)
List(1, 2)	→	List(1, 2)
List(3, 2, 1)	→	List(1, 2, 3)

Logical predicates are a more general form of specification and can be used to constrain an infinite number of inputs and outputs. A logical predicate which specifies the above problem could be

$$\forall i, o. \text{content}(o) == \text{content}(i) \ \&\& \ \text{isSorted}(o),$$

where i, o are the input and output lists of the function respectively, content is a function which computes a set containing a list's elements, and isSorted a boolean function which checks if a list is sorted.

Input-output examples are often easier to write, and also to reason about formally, since they involve only finite sets of values. In contrast, general logical predicates can constrain the output for more inputs, leading more often to the desired synthesis solution. In between these extremes lie *symbolic examples*, that map a possibly infinite family of inputs to corresponding outputs. List sorting could be partially specified with symbolic examples by constraining an infinite set of small lists as follows:

	List()	→	List()
	List(a)	→	List(a)
$\forall a, b, c.$	List(a, b)	→	List(a, b)
	List(c, b, a)	→	List(a, b, c)

Our techniques are general enough to handle all types of specifications, but some behave better for certain types.

In this dissertation, we build on the Leon deductive synthesis framework [KKKS13, Kne16]. Given a synthesis problem, Leon applies on it a series of *deductive synthesis rules* in an effort

to reduce it to other, simpler problems. The successive application of synthesis rules creates a tree of subproblems that Leon explores to solve the initial synthesis problem. At the leaves of the tree lie problems that cannot be profitably decomposed further; these problems are dispatched by so-called *closing rules*. The most important closing rule enumerates and tests candidate programs taken from a program space described by a context-free grammar, until it finds a satisfactory solution.

Contributions. In this dissertation, we make the following contributions:

- We present a set of updates to the Leon deductive synthesis system.
 - We extend the notion of the *path condition* of a synthesis problem. In previous work, a path condition is a predicate on available variables that is known to be true from the context of the problem. In this dissertation, we expand it to include variable bindings and *guides*. Guides are predicates without logical meaning which convey syntactic information about the problem; this information made available for deductive rules to use during synthesis.
- We modify the set of deductive synthesis rules of Leon to utilize such path conditions, and give examples of programs that become solvable with this extension.
- We extend the input language that Leon can handle by adding to it some lightweight object-oriented features and showing how to desugar them to a purely functional language.
 - We introduce *symbolic examples* as a new form of specification. Symbolic examples are input-output examples whose inputs contain abstract values that can be referenced in their output.
- We present *term grammars*, which are context-free grammars or probabilistic context-free grammars whose nonterminal symbols correspond to a program type, optionally decorated with additional information. Term grammars are used in Leon to describe sets of programs that are candidate solutions to a synthesis problem. Those sets can be explored by specific deductive rules to discover a solution to the synthesis problem.

Term grammars were already a part of Leon. In this thesis, we give a more thorough presentation of the different flavors of term grammars we use, and introduce the following novelties:

- We introduce *aspect grammars*. Aspects are a technique to enrich a given term grammar with additional information. This information may encode expected normal forms of terms in the target language or constraints expected to hold in a given context. Examples of these applications are, respectively, encoding that the $+$ operator should have non-zero operands, and that generated terms have to be bound by a specific maximum size.

Each aspect is associated with a transformer between sets of grammar rules. When we attach an aspect to a nonterminal symbol, its production rules are transformed by the aspect's transformer. An advantage of aspects is their modularity: they can be developed separately from the base grammar, and can then be attached to its nonterminals as needed.

- We introduce *generic grammars* containing *generic production rules*, that can be parameterized by type parameters. Generic production rules can express, for instance, that a nonterminal $\text{List}[A]$ corresponding to a type parametric list can expand to either an empty or a nonempty list of the same type ($\text{Nil}[A]$ or $\text{Cons}[A]$). After instantiating its type parameters with concrete types, a generic rule becomes a regular rule and is appended to the other rules of the same nonterminal.

- We describe two term enumeration algorithms that explore a set of programs described by a term grammar to discover a program satisfying the specification of a synthesis problem. Both algorithms are instantiations of CEGIS.

The first is an evolution of the *Symbolic Term Exploration* algorithm presented in previous work on Leon. It explores terms in order of increasing size. We improve the performance of the algorithm by more eagerly eliminating erroneous programs with concrete execution.

The second algorithm is called *Probabilistic Term Enumeration* (PTE) and operates with probabilistic grammars. PTE represents derivations of its input term grammar with a graph, where an edge corresponds to an expansion of a rule of the grammar. It explores this graph in order of descending probability, using the A* best-first search algorithm. We incorporate a number of domain-specific optimizations into PTE and experimentally showcase their usefulness. By using probabilities, best-first search and our optimizations, the algorithm quickly arrives to the programs that are more likely to be correct solutions.

- We describe a procedure that discovers and repairs bugs in functional programs. In the repair problem, we are given a function with correct specification but incorrect implementation, and aim to replace parts of this function with newly synthesized code such that the specification is satisfied.
 - We develop a novel fault localization process that isolates the bug in specific control flow branches of the program. This process uses traces of tests that fail the specification to discover which branches of the program are responsible for the bug.
 - As part of fault localization, we develop a trace (or test) minimization procedure that filters out erroneous traces whose failure can be attributed to a subtrace. This way, we avoid localizing on branches which are themselves correct, but contain a recursive call invoking an erroneous branch.

- After localizing the error, we deploy a variant of synthesis, called *similar term exploration*, to suggest possible fixes. Similar term exploration generates snippets similar to the original erroneous code by inserting to it small variations. It uses synthesis as described above, but with grammar that describes variations to the original program term. This grammar is implemented with an aspect. If similar term exploration fails, we disregard the original snippet and fall back to regular synthesis.

Outline. The rest of this dissertation is organized as follows:

- In Chapter 1, we first present the synthesis problem. We then give an overview of the Leon deductive synthesis system, including its input language, its deductive synthesis rules, and examples of its operation.
- In Chapter 2, we provide an extensive presentation of the flavors of term grammars used to represent program spaces in Leon. Term grammars are used by the term enumeration algorithms of Chapter 3.
- In Chapter 3, we present and evaluate the two term enumeration algorithms used by the deductive rules of Leon described in Chapter 1.
- In Chapter 4, we develop a repair system which localizes bugs in functional programs, then uses the synthesis techniques described in the previous chapters to generate fixes for those bugs.
- In Chapter 5, we present an overview of related work in the field of synthesis and repair.
- Finally, we conclude this dissertation in Chapter 6.

1 Deductive Synthesis and the Leon Framework

In this dissertation, we improve the effectiveness of synthesis techniques through insights into efficient construction and exploration of program spaces. We use the Leon formal methods framework, specifically its deductive synthesis and repair modules, as a framework in which we incorporate our ideas.

This chapter presents a combination of preexisting and original work. Sections 1.2.3, 1.3.1, as well as parts of Section 1.5.6 as indicated, expand on novel work appearing in a previous publication [KKK16] by the author of this dissertation and others. The same is true for sections 1.2.4 and 1.4.3 [KKK15]. The content of Section 1.4.4 first appears in this dissertation. The rest of the chapter presents previous work on synthesis in Leon [KKKS13, Kne16, BKKS13].

1.1 Overview of Leon

Leon is a formal methods framework operating on a functional subset of the Scala language [BKKS13], as well as lightweight imperative and object-oriented extensions (the latter of which is presented in this work). Leon started as a verifier for recursive functional programs [SKK11, VKK15] and evolved into a versatile formal methods system including modules for termination, synthesis [KKKS13], repair [KKK15], and resource bound verification and inference [MK14, MKK17].

The frontend of Leon uses the Scala parser and type checker, then translates Scala ASTs into its own ASTs. Whereas Scala ASTs are mostly syntactic, Leon's ASTs are semantic trees, for instance, they differentiate between different arithmetic expressions, or methods (defined in classes) and functions (defined in objects/modules). Then, a preprocessing phase follows, which desugars the additional imperative and object-oriented language features into the core functional language. Such features include **while**-loops, mutable local variables, method calls, and constrained class hierarchies. Additionally, preprocessing instruments certain expressions with additional verification checks. For instance, Leon adds assertions that check that divisors of divisions are not zero, and that maps are defined on every accessed key.

The frontend is followed by one of the different modules of Leon according to the options chosen, such as termination analysis, verification or synthesis. All modules interact with an SMT-based verifier for recursive functional programs. Specifically in synthesis, the verifier is used to verify the correctness of candidate solutions. For simplicity, we assume that the Leon verifier exposes a single query named `LEONSMT(ϕ)`, that will examine the satisfiability of the logical formula ϕ with three possible results: either `UNSAT`, `UNKNOWN` or `SAT(m)`. The latter also returns a model m that is a mapping from the free variables of ϕ to values such that ϕ is satisfied.

The verification and termination parts of Leon have lately evolved into the Inox [Ino] and Stainless [Sta] projects. Compared to Leon, Inox and Stainless use more modular implementation techniques that allow the different components of the system, including different recognized languages, to be developed separately, without invasive changes to the core of the system. Inox implements the core verification functionalities for recursive higher-order functions and interfaces with the SMT solver, whereas Stainless builds on top of it to implement verification and termination for programs taken from a growing fragment of Scala. It is future work to integrate the synthesis and repair modules of Leon with this new infrastructure.

In this dissertation, we focus on the synthesis and repair modules of Leon. From now on, when we refer to Leon, we usually mean solely its synthesis and repair modules.

In the rest of this chapter, we present illustrative examples of problems that Leon can solve, we introduce the synthesis problem, and outline the techniques we use to solve it. Repair will be discussed in detail in Chapter 4.

1.2 Examples of Synthesis Problems

1.2.1 Distinct Number

We first illustrate the basic features of the Leon synthesis framework through this simple numeric example, that asks for a natural number distinct from two given natural numbers:

```
def distinct(x: BigInt, y: BigInt): BigInt = {  
  require(x ≥ 0 && y ≥ 0)  
  ???[BigInt]  
} ensuring { res ⇒  
  res ≥ 0 && res != x && res != y  
}
```

The `distinct` function has two arguments of type `BigInt`, the type of mathematical (unbounded) integers in Scala. The **require** clause introduces a *precondition* that constrains the inputs of the function to be natural numbers. The **ensuring** clause introduces a *postcondition*, a predicate that constrains the input and desired output of the function. The variable `res` in the postcondition binds to the function's result. `???` represents a typed *synthesis hole* that

the synthesizer has to fill with an implementation. Note that all synthesis constructs are executable: **require** and **ensuring** are regular Scala function defined in the Scala standard library, whereas a synthesis hole can be evaluated with runtime constraint solving [KKKS13, KKS12], although this is not guaranteed to succeed.

After compiling the input file, the synthesizer takes around one second to produce the following output, that is then verified by the Leon verifier:

```
def distinct(x: BigInt, y: BigInt): BigInt = {
  require(x ≥ BigInt(0) && y ≥ BigInt(0))
  x + (y + BigInt(1))
} ensuring { (res : BigInt) ⇒
  res ≥ 0 && res != x && res != y
}
```

1.2.2 Sorted List Insertion and List Sorting

In this next example, we define a synthesis problem with algebraic data types (ADTs).

```
sealed abstract class List[T] {
  def content: Set[T] = this match {
    case Nil() ⇒ Set()
    case Cons(h, t) ⇒ Set(h) ++ t.content
  }
}
case class Cons[T](h: T, t: List[T]) extends List[T]
case class Nil[T]() extends List[T]

def isSorted(list: List[BIGInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) ⇒ x1 < x2 && isSorted(t)
  case _ ⇒ true
}

def insert(in: List[BIGInt], v: BIGInt): List[BIGInt] = {
  require(isSorted(in))
  ???[List[BIGInt]]
} ensuring { (out : List[BIGInt]) ⇒
  (out.content == in.content ++ Set(v)) && isSorted(out)
}
```

We first define a linked list as a polymorphic ADT in Scala. Note that the List defines a method content; this is part of an object-oriented extension for Leon and will be desugared into a function. We then define the notion of (strict) sortedness on such lists (isSorted function).

Chapter 1. Deductive Synthesis and the Leon Framework

Our synthesis task this time is to synthesize a function that inserts an element in the correct position in a sorted list. Note that the precondition and postcondition of insert demand that both the input and output lists be sorted. Since we impose strict sortedness (that is, we use strict inequality to compare elements), the new element should not be inserted if it is already contained in the list.

After approximately 30 seconds, Leon comes up with the following solution:

```
def insert(in : List[BigInt], v : BigInt): List[BigInt] = {  
  require(isSorted(in))  
  in match {  
    case Nil() ⇒ List(v)  
    case Cons(h, t) ⇒  
      val rec = insert(t, v)  
      if (h == v) rec  
      else if (h < v) Cons[BigInt](h, rec)  
      else Cons[BigInt](v, Cons[BigInt](h, t)) }  
} ensuring {  
  (out : List[BigInt]) ⇒  
    out.content == in.content ++ Set[BigInt](v) && isSorted(out) }
```

Leon begins solving the problem by pattern matching on the input variable `in`. In the `Cons` case, it first computes the result of a recursive call to `insert` invoked on the tail of the list, and uses its value in the code following. Although recursive calls can potentially introduce non-termination to the synthesized function, this recursive call does not, as tail is structurally included in the original argument `in`. Given the recursive call, the `Cons` branch is solved with a comparison between the two visible integer variables `head` and `v` that creates three separate branches, each of which is solved individually.

After synthesizing the `insert` function, Leon can use it to synthesize the insertion sort algorithm:

```
def insertionSort(in: List[BigInt]): List[BigInt] = {  
  choose { (out: List[BigInt]) ⇒  
    out.content == in.content && isSorted(out) }  
}
```

// Solution:

```
def insertionSort(in : List[BigInt]): List[BigInt] = {  
  in match {  
    case Nil() ⇒ List[BigInt]()  
    case Cons(h, t) ⇒ insert(insertionSort(t), h) }  
} ensuring { (out : List[BigInt]) ⇒  
  out.content == in.content && isSorted(out)  
}
```

1.2.3 Run-Length Encoding

In the subsequent, more complicated example, we synthesize the run-length encoding of a functional list. A run-length encoding compresses a list by grouping together multiple consecutive appearances of the same element. The returned list consists of pairs (i, e) , where i the number of consecutive appearances of element e . For example, `List(a, a, a, b, c, c)` is encoded as `List((3,a), (1,b), (2,c))`.

We present two separate versions of the problem: In the first version, we specify encode by its inverse function decode, along with a function that constrains legal encodings:

```
def decode[A](l: List[(BigInt, A)]): List[A] = {
  def fill[A](i: BigInt, a: A): List[A] = {
    if (i > 0) a :: fill(i - 1, a)
    else Nil[A]()
  }
  l match {
    case Nil() => Nil[A]()
    case Cons((i, x), xs) =>
      fill(i, x) ++ decode(xs)
  }
}

def legal[A](l: List[(BigInt, A)]): Boolean = l match {
  case Nil() => true
  case Cons((i, _), Nil()) => i > 0
  case Cons((i, x), tl@Cons(_, y, _)) =>
    i > 0 && x != y && legal(tl)
}

def encode[A](l: List[A]): List[(BigInt, A)] = {
  ???[List[(BigInt, A)]]
} ensuring {
  (res: List[(BigInt, A)]) =>
    legal(res) && decode(res) == l
}
```

Leon is able to synthesize the following solution for the problem in around 25 seconds:

```
def encode[A](l : List[A]): List[(BigInt, A)] = {
  l match {
    case Nil() =>
      Nil()
    case Cons(h, t) =>
```

```

encode[A](t) match {
  case Nil() ⇒
    List((BigInt(1), h))
  case Cons(h1 @ (h_1, h_2), t1) ⇒
    if (h == h_2) {
      Cons[(BigInt, A)]((h_1 + BigInt(1), h_2), t1)
    } else {
      Cons[(BigInt, A)]((BigInt(1), h), Cons[(BigInt, A)](h1, t1))
    } } }

```

In the second version of the problem, we specify the desired functionality with *symbolic examples*. Symbolic examples are introduced by the **passes** construct in the postcondition of a function. An acceptable synthesis solution must satisfy every provided symbolic example for every instantiation of its symbolic values. We discuss symbolic examples in more detail in Section 1.4.3.

```

def encode[A](l: List[A]): List[(BigInt, A)] = {
  ???[List[(BigInt, A)]]
} ensuring {
  (res: List[(BigInt, A)]) ⇒
    (l, res) passes {
      case Nil() ⇒ Nil()
      case Cons(a, Nil()) ⇒
        List((1,a))
      case Cons(a, Cons(b, Nil())) if a == b ⇒
        List((2,a))
      case Cons(a, Cons(b, Cons(c, Nil()))) if a == b && a == c ⇒
        List((3,a))
      case Cons(a, Cons(b, Cons(c, Nil()))) if a == b && a != c ⇒
        List((2,a), (1,c))
      case Cons(a, Cons(b, Cons(c, Nil()))) if a != b && b == c ⇒
        List((1,a), (2,b))
      case Cons(a, Cons(b, Nil())) if a != b ⇒
        List((1,a), (1,b)) } }

```

Leon comes up with the same solution as the other variation of the problem, in a time varying from 12 to 41 seconds depending on its configuration.

This benchmark was not synthesizable with previous versions of Leon. Improvements that made it possible include a new approach to introducing recursive function calls (Section 1.5.6), the optimization of the existing term enumeration algorithm in Leon (Section 3.2), as well as the development of a new such algorithm (Section 3.3).

1.2.4 Repairing Heap Merging

We conclude this section by displaying an example of repair with Leon. We will discuss repair further in Chapter 4.

The following example is a fragment from a benchmark which implements leftist max-heaps. The full benchmark can be found in Appendix C.2. A bug has crept in the indicated line of the heap merge algorithm: The correct heap *l2* has been swapped for *l1*. Leon's repair module is able to locate and fix this error within a few seconds.

```
sealed abstract class Heap {
  ...
  def content : Set[BigInt] = this match {
    case Leaf() => Set[BigInt]()
    case Node(v,l,r) => l.content ++ Set(v) ++ r.content }
}
case class Leaf() extends Heap
case class Node(value: BigInt, left: Heap, right: Heap) extends Heap

def hasHeapProperty(h : Heap) : Boolean = ...
def hasLeftistProperty(h: Heap) : Boolean = ...
def heapSize(t: Heap): BigInt = ...

private def makeN(value: BigInt, left: Heap, right: Heap) : Heap = {
  require(hasLeftistProperty(left) && hasLeftistProperty(right))
  if(left.rank ≥ right.rank) Node(value, left, right)
  else Node(value, right, left)
} ensuring { res => hasLeftistProperty(res) }

private def merge(h1: Heap, h2: Heap) : Heap = {
  require(
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&
    hasHeapProperty(h1) && hasHeapProperty(h2) )
  (h1,h2) match {
    case (Leaf(), _) => h2
    case (_, Leaf()) => h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) =>
      if(v1 ≥ v2) makeN(v1, l1, merge(r1, h2))
      else makeN(v2, l1, merge(h1, r2)) } // Bug: l1 instead of l2
  } ensuring { res =>
    hasLeftistProperty(res) && hasHeapProperty(res) &&
    heapSize(h1) + heapSize(h2) == heapSize(res) &&
    h1.content ++ h2.content == res.content }
```

1.3 The Synthesis Problem

In this section, we formally define the synthesis problem.

Let $\phi = \phi(\bar{a}, x)$ be the *specification* of a programming task, specifying a relation between a tuple of input parameters \bar{a} and an output variable x . ϕ can be given as a logical predicate, a set of input-output examples, a reference implementation, or a combination thereof. Also, let Π be the *path condition* for the problem, a predicate on the input parameters satisfied by hypothesis. In Section 1.3.1 we will extend the notion of path condition. Also, let T be a term in the target language. The free variables of both Π and T have to be subsets of \bar{a} .

The *synthesis problem* asks for a constructive solution for the formula

$$\exists T. \forall \bar{a}. [\Pi \rightarrow \phi[x \mapsto T]] \quad (1.1)$$

In other words, we want to find an expression T such that, when we replace every occurrence of the output variable x in the specification ϕ by T , the formula $\Pi \rightarrow \phi$ becomes valid.

We will write such a synthesis problem for short as

$$\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket. \quad (1.2)$$

If we want to expand the input parameter tuple, we can write

$$\llbracket a_1, a_2, \dots, a_n \langle \Pi \triangleright \phi \rangle x \rrbracket \text{ or } \llbracket \langle \Pi \triangleright \phi \rangle x \rrbracket,$$

depending on whether the tuple is nonempty or not.

A solution to the synthesis problem is a pair $\langle P \mid T \rangle$, where P is a precondition that constrains the domain of the solution, and T is the synthesized term. If $\langle P \mid T \rangle$ is a solution to Equation 1.2, we will write

$$\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle \quad (1.3)$$

For example, take the synthesis problem with an input parameter a , where we are searching for an integer whose double is a . The synthesizer could come up with the solution $a/2$, which is only valid when a is even:

$$\llbracket a \langle \text{true} \triangleright 2 \cdot x == a \rangle x \rrbracket \vdash \langle a \% 2 == 0 \mid a/2 \rangle$$

Of course, we would like P to be as weak as possible, ideally *true*. In fact, the version of Leon presented in this dissertation solves all of our benchmarks using only rules that return *true* as a precondition.

1.3.1 Path Conditions

The notion of *path condition* was first defined by King in the context of symbolic execution [Kin76] to capture information about input variables that is known to hold in a specific control flow path. Similarly, in synthesis we use path conditions to capture known information about the inputs of a synthesis problem. In this dissertation, we use an extended notion of path condition compared to King.

A path condition in this dissertation is a conjunction of any number of three types of clauses, as described by the following grammar:

$$\begin{aligned}\Pi &::= \mathbf{true} \mid e \wedge \Pi \mid w(e) \wedge \Pi \mid (v \leftarrow e) \wedge \Pi \\ v &::= \text{a Scala identifier} \\ e &::= \text{a Scala expression that type-checks in its context} \\ w &::= \text{a finite set of witness symbols}\end{aligned}$$

The semantics of these clauses (excluding the trivial path condition **true**) is respectively the following:

- A predicate on the input parameters that is satisfied by hypothesis.
- An instance of a *witness*. A witness is a predicate with no logical meaning (equivalent to **true**), whose purpose is to track syntactic information. This information is used during both synthesis and repair. For example, \odot is a witness pronounced *guide*. Writing $\odot[f(a)] \wedge \Pi$ denotes a path condition containing the guide $f(a)$.
- A binding of a fresh variable to an expression. $(v \leftarrow e) \wedge \Pi$ binds v to e and appends this information to the path condition Π . As an example, when introducing a recursive call invoking the function under synthesis, we bind its value to a variable. The synthesizer uses bound variables similarly to input parameters in \bar{a} , though the two categories are handled differently in some ways.

We write $\text{bound}(\Pi)$ to refer to variables bound in Π .

1.4 Input Language

We now describe the input language accepted by Leon. This language is a subset of Scala, consisting of a core functional language, as well as some additional extensions that get desugared away after parsing. As mentioned before, we use the Scala compiler (Scala version 2.11.8) to initially parse and type check input code.

The core functional language of Leon is called PureScala [BKKS13]. PureScala corresponds to an ML-like subset of Scala. It contains pure recursive functions operating on integral types, booleans, and ADTs. Apart from standard functional language features, PureScala provides formal contracts and synthesis constructs. Leon internally analyzes and synthesizes

programs only in PureScala; all other language components get desugared before invoking the synthesizer. Also, the synthesis component of Leon is constrained to the first-order fragment of PureScala.

In Section 1.4.4 we introduce a lightweight object-oriented extension to PureScala, which introduces a simple form of class hierarchies with methods and fields. Leon also accepts another extension to PureScala called XLang [BKKS13, Bla17], which introduces a set of imperative features that are also desugared into functional code. XLang is not described in this dissertation.

1.4.1 Syntax of PureScala

In Figure 1.1 we give the grammar describing the syntax of PureScala [BKKS13]. Nonterminal symbols are shown in *italic* font, while terminal symbols are shown in **bold** font. The syntax is approximate, as we do not account for operator precedence, or Scala syntax nuisances involving necessary braces, etc.

A program is a sequence of compilation units, each of which contains a package declaration, a sequence of imports and a sequence of object (module) or class definitions. Modules contain functions or classes. Classes define ADTs: abstract classes define types and case classes define ADT constructors. Case classes can only extend abstract classes. A case class that does not extend another class defines a type as well as a constructor. Functions can have pre- and postconditions. Expressions describe, among others, errors –which must be unreachable– and asserts –whose argument must be provably true–, literals, variables and local variable definitions, function (lambda) calls and definitions, function (**def**-definition) calls, **match** and **passes** constructs –the latter of which introduces symbolic examples–, set and map operations, type checks and casts, field accesses, and finally synthesis constructs. We emphasize again that the synthesis component of Leon is constrained to the first-order fragment of PureScala.

Type checks (`isInstanceOf`) are acceptable in a purely functional context because they are equivalent to a simple pattern matching-based check. Type casts (`asInstanceOf`) are only acceptable if they can be proven to be safe. When invoked for verification, Leon will emit proof obligations for type casts, similarly to those for asserts and errors. Field accesses are safe because they are only allowed by the Scala type checker if they are well-typed. **passes** introduces symbolic examples and is only allowed as a top-level conjunct in a postcondition.

1.4.2 Synthesis Constructs

The task of the synthesizer is to substitute each nondeterministic choice operator **choose** in the program with an expression that matches its specification. The second synthesis construct, namely the synthesis hole `???`, is syntactic sugar for **choose**. Intuitively, **choose** defines its specification locally, whereas the specification for hole is computed from the specifications of the function.


```

program ::= unit+
unit ::= packageDef? import* ocDef*
packageDef ::= package < id . >* id
import ::= import < id . >* < id | _ >
ocDef ::= objectDef | classDef
objectDef ::= object id { < fDef | classDef >* }
fDef ::= funDef
classDef ::= abstract class id tparams
          | case class id tparams ( < param < , param >* >? ) < extends id tparams >?
funDef ::= def id tparams ( < param < , param >* >? ) : type = {
          < require( expr ) >?
          expr
          } < ensuring { id ⇒ expr } >?
param ::= id : type
expr ::= error( stringLit ) | assert( expr, stringLit ); expr
      | id | literal | ( expr ) | val id = expr ; expr
      | expr ( < expr < , expr >* >? ) | ( < id < , id >* >? ) ⇒ expr
      | qid [ < type < , type >* > ]? ( < expr < , expr >* >? )
      | if ( expr ) { expr } else { expr }
      | expr < match | passes > { case+ }
      | expr binop expr | unop expr
      | ( expr , expr < , expr >* ) | expr._n
      | Set[ type ] ( < expr < , expr >* >? ) | expr.size
      | expr.contains( expr ) | expr.subsetOf( expr )
      | Map[ type, type ] ( < expr < , expr >* >? ) | expr.isDefinedAt( expr )
      | expr.isInstanceOf[ type ] | expr.asInstanceOf[ type ]
      | expr.id | choose ( expr ) | ??? [ type ]
binOp ::= == | && | || | → | + | - | * | / | %
      | ≤ | ≥ | < | > | & | || | << | >> | >>> | ++ | --
unOp ::= - | !
n ::= 1 | 2 | ...
type ::= BigInt | Int | Boolean | Set[ type ]
      | Map[ type, type ]
      | qid < [ type < , type >* > ]?
case ::= pattern < if expr >? ⇒ expr
pattern ::= id | < id @ >? _ | < id @ >? literal
          | < id @ >? ( pattern < , pattern >* )
          | < id @ >? qid ( < pattern < , pattern >* >? )
literal ::= 0 | 1 | ... | BigInt(0) | BigInt(1) | ...
          | true | false | () | 'a' | 'b' | ... | stringLit
stringLit ::= "stringChar*"
tparams ::= < [ id < , id >* ]?
qid ::= < id . >? id

```

Figure 1.1 – Syntax of PureScala

<pre>def f(...): T = { require(pre) C(???₁, ???₂, ..., ???_n) } ensuring (x ⇒ post)</pre>	\rightarrow	<pre>def f(...): T = { require(pre) val (h₁, h₂, ..., h_n) = choose { (h'₁, h'₂, ..., h'_n) ⇒ val x = C(h'₁, h'₂, ..., h'_n) post } C(h₁, h₂, ..., h_n) } ensuring (x ⇒ post)</pre>
---	---------------	---

Figure 1.2 – Desugaring of synthesis holes

Figure 1.2 presents the desugaring of a function body with holes into one with **choose** constructs. In the figure, C is the body of the function defined as an expression context containing holes $???_1, ???_2, \dots, ???_n$.

1.4.3 Symbolic Examples

Writing a complete logical specification for a synthesis problem, if indeed such a meaningful specification exists, can often be a tedious task. In these cases, it is often helpful to express intent through input-output examples; however, input-output examples are often incomplete specification and result in ambiguity in the generated solutions.

Consider trying to specify a function that computes the size of a functional list. On the one hand, we do not know a specification simpler than the function's implementation. On the other hand, we would need numerous input-output examples to adequately constrain the generated solutions.

For instance, given the input-output pairs $\{\text{Nil}() \rightarrow 0, \text{Cons}(0, \text{Nil}()) \rightarrow 1\}$, a satisfying solution is

```
def size(l: List): BigInt = l match {
  case Nil() ⇒ 0
  case Cons(h, t) ⇒ 1 + h
}
```

One fundamental problem here is that there is no way to communicate that every list of the same size should be mapped to the same output. We could keep adding examples to the specification until we constrain the output enough to obtain the correct solution, but this would be tedious.

To solve this problem, we propose the use of *symbolic examples* [KKK15]. Symbolic examples are a form of specification between regular (concrete) input-output examples and full logical specification. Symbolic examples can contain abstract values as part of their input, which can be referenced in the output.

<pre> ((a₁, ..., a_n), x) passes { case p₁ ⇒ v₁ ... case p_k ⇒ v_k } </pre>	\triangleq	<pre> val in = (a₁, ..., a_n) val x' = x in match { case p₁ ⇒ x' == v₁ ... case p_k ⇒ x' == v_k case _ ⇒ true } </pre>
---	--------------	---

Figure 1.3 – Definition of **passes** in Scala

In the case of the size function, a set of symbolic examples could be

(1) $\text{Nil}() \rightarrow 0$, (2) $\text{Cons}(x, \text{Nil}()) \rightarrow 1$ and (3) $\text{Cons}(x, \text{Cons}(y, \text{Nil}())) \rightarrow 2$. To define symbolic examples in Scala, we introduce a built-in construct we call **passes**, that uses Scala's pattern matching and partial functions:

```

def size(l: List): BigInt = ??? ensuring { res ⇒
  (l, res) passes {
    case Nil() ⇒ 0
    case Cons(_, Nil()) ⇒ 1
    case Cons(_, Cons(_, Nil())) ⇒ 2 } }

```

passes is executable and defined in terms of default Scala expressions as shown in Figure 1.3.

We can use wildcard patterns to avoid naming values we will not use in the output. Leon enforces that the left-hand side of **passes** is a pair consisting of a tuple of input variables and the output variable. In the special case where we have no variables or wildcards in the patterns of **passes**, it corresponds to common input-output examples.

1.4.4 Object-Oriented Extension

In Figure 1.4 we give the syntax for an object-oriented syntax extension for PureScala. As this fragment is desugared away by the frontend of Leon, before any other modules apply, it can be used in programs directed to any Leon module, notably synthesis, repair and verification. It was motivated by a desire to handle a larger fragment of the Scala language to give developers the opportunity to write more natural Scala code. An example of the use of this extension is the Heap type defined in Appendix C.2.

The features introduced in this extension are, in order of appearance in Figure 1.4, fields as an alternative to functions, extended class definitions which can contain field and method definitions and where abstract classes can extend other abstract classes, field definitions, **this** object references, class field dereferences, and method invocations. The extended class definitions allow for type hierarchies to resemble trees, whose leaves are case classes and

```

fDef ::= funDef | fieldDef
classDef ::= abstract class id tparams < extends id tparams >? < { fDef* } >?
           | case class id tparams ( < param < , param >* >? ) < extends id tparams >?
           < { fDef* } >?
fieldDef ::= < lazy >? val id : type = {
           < require( expr ) >?
           expr
           } < ensuring { id ⇒ expr } >?
expr ::= ... | this | expr . id
           | expr . id < [ type < , type >* ] >? ( < expr < , expr >* >? )

```

Figure 1.4 – Syntax modifications for object-oriented extensions

inner nodes are abstract classes. When we refer to “field definitions” in classes, we mean **val** definitions in the body of a class and not constructor arguments (that correspond to ADT fields).

This small additional subset gets desugared into PureScala in a preprocessing step in Leon, before synthesis is invoked. During this desugaring step, the tree-like type hierarchy described above is reduced to ADT definitions. Expressions involving type checks and casts are also transformed to correspond to the new type structure. Additionally, method definitions and calls are reduced to function definitions and calls. Fields are transformed similar to methods and are handled in Leon as functions with zero parameters.

The process described bellow is capable of handling fields that are implemented or overridden in subclasses as constructor arguments, as indicated in the following class definitions:

```

abstract class A { val f: BigInt = 0 }
case class B(override val f: BigInt) extends A

```

We next detail the operation of the desugaring of the object-oriented features. In the actual implementation, some of these changes are postponed to later phases of Leon, but we present them here as one coherent process. As a running example, consider the program of Figure 1.5.

- (1) The root of each defined type hierarchy is mapped to a type definition in PureScala. The leaves (case classes) of such a hierarchy are mapped to constructor definitions. Intermediate abstract classes in the type hierarchy are not maintained. In the example of Figure 1.5, note that the original type hierarchy has been desugared into a type named A with constructors B, D and E.

Case classes that are also roots of a type hierarchy are maintained and correspond both to a type and a constructor definition, as mentioned Section 1.4.1.

- (2) Each class field/method definition *m*, including those in subclasses, is mapped to a

```

1  // Original
2  abstract class A {
3    val fld: BigInt
4    def fun(p: BigInt): BigInt }
5  case class B(override val fld: BigInt) extends A {
6    def fun(p: BigInt): BigInt = 0 }
7  abstract class C extends A {
8    def fun(p: BigInt): BigInt = this.fld + p }
9  case class D() extends C {
10   override val fld: BigInt = 0
11   override def fun(p: BigInt): BigInt = this.fld + p + 1 }
12 case class E() extends C {
13   override val fld: BigInt = 42
14   def anotherFun(p1: BigInt) = fld + p1 + 1 }
15
16 // Desugared
17 abstract class A
18 case class B(fld : BigInt) extends A
19 case class D() extends A
20 case class E() extends A
21
22 def fld(thiss : A): BigInt = thiss match {
23   case b @ B(fld1) =>
24     fld1
25   case d @ D() =>
26     BigInt(0)
27   case e @ E() =>
28     BigInt(42) }
29
30 def fun(thiss : A, p : BigInt): BigInt = thiss match {
31   case b @ B(fld) =>
32     BigInt(0)
33   case d @ D() =>
34     fld(d) + p + BigInt(1)
35   case e @ E() =>
36     fld(e) + p }
37
38 def anotherFun(thiss : A, p1 : BigInt): BigInt = {
39   require(thiss.isInstanceOf[E])
40   fld(thiss.asInstanceOf[E]) + p1 + BigInt(1) }

```

Figure 1.5 – Desugaring of an object-oriented program

function f with an additional argument a_0 which represents the receiver object (**this**). The type of a_0 is always the root of the type hierarchy. The type parameters of f are the type parameters of m plus the type parameters of the class C that defines m . In the example, note the signature of `fun`, `fld` and `anotherFun`, the latter of which was originally defined in subclass `E`.

The f function is assigned a body composed from the body of m in all subclasses. This body contains a top-level match expression which emulates dynamic dispatch by dynamically checking the type of a_0 and invoking the appropriate version of the method. Observe the bodies of `fun` and `fld` in the example.

The body of f is built bottom-up starting from the leaves of the type hierarchy. We will label T the desugaring transformation that builds up f . $T(m, C)$ takes as input the method m being transformed and the class C currently being analyzed, and returns a pair of $(cases, total)$. $cases$ is a tuple of match-cases, containing one match-case for each subclass of C , including C itself. These cases each give the result of the function if a_0 is found at runtime to be of the corresponding class. $total$ is a boolean value signifying whether $cases$ totally covers the definition of $expr$ for C and all its subclasses. The functionality of $total$ will become clear below.

The effect of T depends on the kind of the current class C being analyzed.

If C is a case class,

- If C does not define m , i.e., m is defined in a superclass or on a different branch of the type hierarchy, then $T(m, C) = ((), false)$. In the example, this will happen while analyzing `fun` for class `E`.
- If C defines m as a method/field, then $T(m, C) = (case\ C(f'_1, \dots, f'_n) \Rightarrow m_C, true)$, where f'_i are identifiers corresponding to the fields f_i of C , and m_C is the body of m in C with the following modifications:
 - **this** is substituted by a_0 .
 - **this**. f_i or, equivalently, a reference to field f_i , is substituted by f'_i (the respective binder in the pattern).

In the example, this happens when transforming `fun` for `D`.

- If C defines m as an ADT constructor field f , then $T(m, C) = (case\ C(_, \dots, f, \dots, _) \Rightarrow f, true)$. In the example, this happens when transforming `fld` for `B`.

If C is an abstract class, let \bar{t} be the tuple of the outputs of T when invoked on the subclasses of C .

- If $total = true$ for every $t \in \bar{t}$, then the definition of m in C , even if present, gets overridden in every subclass, so the current definition is irrelevant. Therefore, $T(m, C) = (cases, true)$, where $cases$ are all the cases in \bar{t} . In the example, this happens when desugaring `fun` for `A`.

- If $total = false$ for some $t \in \bar{t}$, and m is not defined in C , then $T(m, C) = (cases, false)$ with $cases$ defined as above. In the example, this happens when desugaring `anotherFun` for C .
 - Finally, if $total = false$ and C defines m , this definition will be used by default for subclasses that do not override m . In this case, $T(m, C) = (cases, true)$, where $cases$ are all the cases in \bar{t} with the additional appended cases (case $b : C_{sub} \Rightarrow m_{C_{sub}}$) for every C_{sub} subclass of C that does not define an element in the tuple \bar{t} . Here, b is a fresh identifier, and $m_{C_{sub}}$ is the body of m in C_{sub} with any references to **this** substituted for b . In the example, this happens when desugaring `fun` for C .
- (3) A precondition is added to each generated function f constraining it to the classes it was originally defined in. In the example, this is required only in the definition of `anotherFun`.
- (4) Each type check `isInstanceOf[A]` for an intermediate abstract class A is transformed to a disjunction of type checks (`isInstanceOf[C1] || ... || isInstanceOf[Ck]`), where C_i are the case-class subclasses of A .
- Each type cast with `asInstanceOf[A]` to an abstract class A is removed. Note that it is safe to remove those casts. If A is a class hierarchy root, then the cast is redundant. If A is an intermediate class, then the cast could only serve to access a field/method defined in subclass A . But this definition has now been mapped to a function defined on the type hierarchy root, which makes the type cast redundant. Note that the discrepancy between the domains of the original method/field and the resulting function is compensated by the precondition added to said function.
- (5) Invocations of the form $o.m(\bar{a})$ are substituted by function calls of the form $f(o, \bar{a})$. Field dereferences of the form $o.m$ are substituted by function calls of the form $f(o)$. Finally, generated functions are added to the program, and method/field definitions in classes are removed.

1.5 Deductive Synthesis

In this section, we discuss the details of how Leon employs *deductive synthesis* to solve synthesis problems. Given a synthesis problem, Leon applies on it a series of deductive synthesis rules in an effort to either solve it outright, or reduce it to other, simpler problems. The successive application of synthesis rules creates a directed acyclic search graph whose source is the initial synthesis problem. The system explores this graph to find solutions to the synthesis problem. Each node of the graph corresponds either to a synthesis problem or a *rule instantiation*, the latter of which is explained in the next section.

1.5.1 Rule Instantiation

Given a synthesis problem, Leon tries to *instantiate* on it all available rules. A rule instantiation suggests a way to decompose the problem. Some rules cannot be instantiated on a given problem, and some can be instantiated in multiple distinct ways, each of which results in a different decomposition.

As an example, suppose we are given some synthesis problem

$$\llbracket a, b \langle \text{true} \triangleright \phi \rangle x \rrbracket \quad (1.4)$$

where a, b : `BigInt`. An applicable rule would be `INEQUALITY SPLIT – INPUT`, that breaks down the problem to three subproblems based on the result of the comparison between two integral values. In fact, three separate instantiations of this rule would be available: “`INEQUALITY SPLIT – INPUT` between a and 0”, “`INEQUALITY SPLIT – INPUT` between b and 0” and “`INEQUALITY SPLIT – INPUT` between a and b ”. Each one suggests a different decomposition of the synthesis problem.

For each applicable instantiation, we create a node in the synthesis search graph, and we connect those nodes with edges from the node corresponding to the problem. If one of the rule instantiations results in a solution, the synthesis problem is solved. Therefore problem nodes constitute so-called *OR-nodes* of the graph.

1.5.2 Application of a Rule Instantiation

Each rule instantiation applied to the initial problem returns a tuple of subproblems whose solutions, if found, can be combined into a solution to the initial problem. If one of the subproblems fails to produce a solution, the whole rule application fails. In the example of Equation 1.4, if we apply the rule instantiation “`INEQUALITY SPLIT – INPUT` between a and 0”, the generated subproblems would be $\llbracket a, b \langle a < 0 \triangleright \phi \rangle x \rrbracket$, $\llbracket a, b \langle a > 0 \triangleright \phi \rangle x \rrbracket$ and $\llbracket b \langle \text{true} \triangleright \phi[a \mapsto 0] \rangle x \rrbracket$.

When applying a rule instantiation, we create a new node in the synthesis search graph for each subproblem, and connect those nodes with edges from the rule instantiation. Since all subproblems must be solved for the instantiation to return a solution, rule instantiations constitute so-called *AND-nodes* of the search graph.

In the case where the set of subproblems generated by a rule application is nonempty, the rule is called a *decomposition rule*. In the case where the set is empty, the problem is called a *closing rule*: it will immediately either return a solution to the problem or fail. In theory, a rule could manifest either as a decomposition or closing rule based on the number of generated subproblems in a specific application, but in practice, each rule in Leon always abides to one of the two categories. An application generating a single subproblem can be seen as a simplification rule.

1.5.3 Solution Composition

If all subproblems of a rule instantiation successfully return a solution, the partial solutions are composed to produce a solution to the initial problem. Each rule provides its own composition formula.

In the example of the previous sections, the rule `INEQUALITY SPLIT – INPUT` got instantiated on the problem $\llbracket a, b \langle \text{true} \triangleright \phi \rangle x \rrbracket$. From this instantiation, three subproblems were generated, namely $\llbracket a, b \langle a < 0 \triangleright \phi \rangle x \rrbracket$, $\llbracket a, b \langle a > 0 \triangleright \phi \rangle x \rrbracket$ and $\llbracket b \langle \text{true} \triangleright \phi[a \mapsto 0] \rangle x \rrbracket$. Assume those problems return respectively $\langle P_1 \mid T_1 \rangle$, $\langle P_2 \mid T_2 \rangle$ and $\langle P_3 \mid T_3 \rangle$. Then, according to the composition formula of the rule, the solution to the initial problem would be

$$\langle (a < 0 \wedge P_1) \vee (a > 0 \wedge P_2) \vee (a = 0 \wedge P_3) \mid \text{if } (a < 0) \{T_1\} \text{ else } \{\text{if } (a > 0) \{T_2\} \text{ else } \{T_3\}\} \rangle.$$

This example is summarized in formal notation in Figure 1.7.

In the case of a closing rule, the composition trivially returns the solution produced by the rule.

Note that, even when all the solutions to the subproblems are not known yet, we can construct a partial program according to the composition formulas of the instantiated rules, where the unknown solutions have been substituted for synthesis holes. Later, when we find the solution to a subproblem, we can substitute it in the search graph in place of the corresponding hole. At any given point, the partial program represents the information about the solution that has been discovered so far. This information is useful during synthesis, for instance, during term exploration explained in Chapter 3.

1.5.4 Normalizing Rules

Some rules are labeled as *normalizing* and take priority over other rules. The idea is that the consecutive application of all normalizing rules applicable to a problem will transform it to a normal form, making the application of the other rules more effective and predictable. Every normalizing rule generates exactly one subproblem. An example of a normalizing rule is `DETUPLE INPUT`, which breaks down an input parameter of a tuple type to its elements.

1.5.5 Synthesis Rules as Inference Rules

We can summarize the effect of a synthesis rule on a problem with an inference rule. The conclusion of the rule is a synthesis problem along with its computed solution, whereas the premises of the rule are any logical formulas. Often the premises will be the solutions to other problems, which are the subproblems generated by the rule.

The rule in Figure 1.6 can be read as: “The synthesis problem (a) can be decomposed into subproblems (b) and, given the solutions (c) of the subproblems, can be solved with (d)”. As a

$$\frac{\boxed{\llbracket \bar{a}_1 \langle \Pi_1 \triangleright \phi_1 \rangle x_1 \rrbracket}_{(b)} \vdash \boxed{\langle P_1 \mid T_1 \rangle}_{(c)} \quad \boxed{\llbracket \bar{a}_2 \langle \Pi_2 \triangleright \phi_2 \rangle x_2 \rrbracket}_{(b)} \vdash \boxed{\langle P_2 \mid T_2 \rangle}_{(c)}}{\boxed{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket}_{(a)} \vdash \boxed{\langle P \mid T \rangle}_{(d)}}$$

Figure 1.6 – Generic synthesis rule as inference rule

INEQUALITY SPLIT – INPUT

$$\frac{\begin{array}{l} \boxed{a, b \langle a < 0 \triangleright \phi \rangle x} \vdash \langle P_1 \mid T_1 \rangle \quad \boxed{a, b \langle a > 0 \triangleright \phi \rangle x} \vdash \langle P_2 \mid T_2 \rangle \\ \boxed{b \langle \text{true} \triangleright \phi[a \mapsto 0] \rangle x} \vdash \langle P_3 \mid T_3 \rangle \end{array}}{\begin{array}{l} \boxed{a, b \langle \text{true} \triangleright \phi \rangle x} \vdash \\ \langle (a < 0 \wedge P_1) \vee (a > 0 \wedge P_2) \vee (a = 0 \wedge P_3) \mid \text{if } (a < 0) \{T_1\} \text{ else } \{\text{if } (a > 0) \{T_2\} \text{ else } \{T_3\}\} \rangle \end{array}}$$

Figure 1.7 – Composition example of Section 1.5.3

concrete example, in Figure 1.7 we give the decomposition of the example of Section 1.5.3.

1.5.6 Definition of Synthesis Rules

In this section, we present all synthesis rules that are used by the Leon framework. They are summarized in figures 1.8 to 1.11. More specifically, Figure 1.8 defines Leon’s normalizing rules, figures 1.9 and 1.10 describe the so called *splitting* rules, that decompose a problem based on pattern matching or the result of some boolean test, and finally, Figure 1.11 gives a sketch of a term exploration rule, which is the main closing rule of Leon. Leon transforms and filters user-provided and generated input-output examples along rule instantiations; this effect is not displayed in rule definitions for reasons of readability.

All rules except INTRODUCE RECURSIVE CALLS were already introduced in previous work on Leon. In this dissertation, we modify some of them to handle bound variables, as explained below.

Ground. The GROUND rule is a normalizing rule that is invoked whenever there are no input variables. In this case, the synthesis problem is reduced to an existential query that can be directly dispatched to the Leon solver. If the solver returns SAT(m), the solution is the value of m for the output variable x . If it returns UNSAT, there is no solution to the problem and the rule does not apply. Note that, since GROUND is a normalizing rule, if it is applicable to a given problem, it will be the only applicable rule. Therefore its failure implies that the problem is not solvable.

$\frac{\text{GROUND} \quad \text{LEONSMT}(\phi) = \text{SAT}(m)}{\llbracket \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle \text{true} \mid m(x) \rangle}$	
$\frac{\text{INTRODUCE RECURSIVE CALLS} \quad \llbracket \bar{a} \langle \Pi \wedge (\text{rec} \leftarrow f(e_1, \dots, e'_i, \dots, e_n)) \wedge \Downarrow[f(e_1, \dots, e'_i, \dots, e_n)] \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle \quad e'_i \in \text{argsSmaller}(e_i, \Pi)}{\llbracket \bar{a} \langle \Downarrow[f(e_1, \dots, e_i, \dots, e_n)] \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle}$	
$\frac{\text{UNUSED INPUT} \quad a \in \bar{a} \quad a \notin FV(\Pi) \quad a \notin FV(\phi) \quad \llbracket \bar{a} \setminus a \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle}$	
$\frac{\text{ONE-POINT} \quad x \notin FV(e) \quad e \text{ does not contain recursive calls}}{\llbracket \bar{a} \langle \Pi \triangleright x == e \rangle x \rrbracket \vdash \langle \text{true} \mid e \rangle}$	$\frac{\text{UNCONSTRAINED OUTPUT} \quad x \notin FV(\phi) \quad x : T \quad e = sv[T]}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle \text{true} \mid e \rangle}$
$\frac{\text{DETUPLE INPUT - TUPLE} \quad a \in \bar{a} \quad a : (T_1, \dots, T_n) \quad a_1 : T_1, \dots, a_n : T_n \text{ are fresh variables} \quad \llbracket (\bar{a} \setminus a) \cup a_1 \dots a_n \langle \Pi[a \mapsto (a_1, \dots, a_n)] \triangleright \phi[a \mapsto (a_1, \dots, a_n)] \rangle x \rrbracket \vdash \langle P \mid T \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle \text{val } (a_1, \dots, a_n) = a; P \mid \text{val } (a_1, \dots, a_n) = a; T \rangle}$	
$\frac{\text{DETUPLE INPUT - CASE CLASS} \quad a \in \bar{a} \quad a : C \quad C(f_1 : T_1, \dots, f_n : T_n) \text{ is a case class constructor} \quad a_1 : T_1, \dots, a_n : T_n \text{ are fresh variables} \quad \llbracket (\bar{a} \setminus a) \cup a_1 \dots a_n \langle \Pi[a \mapsto C(a_1, \dots, a_n)] \triangleright \phi[a \mapsto C(a_1, \dots, a_n)] \rangle x \rrbracket \vdash \langle P \mid T \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle \text{val } C(a_1, \dots, a_n) = a; P \mid \text{val } C(a_1, \dots, a_n) = a; T \rangle}$	
$\frac{\text{DETUPLE BOUND VARIABLE - TUPLE} \quad a \in \text{bound}(\Pi) \quad a : (T_1, \dots, T_n) \quad a_1 : T_1, \dots, a_n : T_n \text{ are fresh variables} \quad \llbracket \bar{a} \langle \Pi \wedge a_1 \leftarrow a._1 \wedge \dots \wedge a_n \leftarrow a._n \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle \text{val } (a_1, \dots, a_n) = a; P \mid \text{val } (a_1, \dots, a_n) = a; T \rangle}$	
$\frac{\text{DETUPLE BOUND VARIABLE - CASE CLASS} \quad a \in \text{bound}(\Pi) \quad a : C \quad C(f_1 : T_1, \dots, f_n : T_n) \text{ is a case class constructor} \quad a_1 : T_1, \dots, a_n : T_n \text{ are fresh variables} \quad \llbracket \bar{a} \langle \Pi \wedge a_1 \leftarrow a.f_1 \wedge \dots \wedge a_n \leftarrow a.f_n \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle \text{val } C(a_1, \dots, a_n) = a; P \mid \text{val } C(a_1, \dots, a_n) = a; T \rangle}$	

Figure 1.8 – Normalizing synthesis rules

INEQUALITY SPLIT – INPUT

$$\begin{array}{c}
 a_1 \in \bar{a} \quad a_2 \in \bar{a} \cup \text{bound}(\Pi) \cup \{0\} \quad a_1, a_2 : T \quad T \in \{\text{Int}, \text{BigInt}\} \\
 \llbracket \bar{a} \setminus a_1 \langle \Pi[a_1 \mapsto a_2] \triangleright \phi[a_1 \mapsto a_2] \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \\
 \llbracket \bar{a} \langle \Pi \wedge a_1 > a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \quad \llbracket \bar{a} \langle \Pi \wedge a_1 < a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_3 \mid T_3 \rangle \\
 P' = a_1 == a_2 \wedge P_1 \vee a_1 > a_2 \wedge P_2 \vee a_1 < a_2 \wedge P_3 \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P' \mid \text{if } (a_1 == a_2) T_1 \text{ else if } (a_1 > a_2) T_2 \text{ else } T_3 \rangle
 \end{array}$$

INEQUALITY SPLIT – BOUND

$$\begin{array}{c}
 a_1 \in \text{bound}(\Pi) \quad a_2 \in \text{bound}(\Pi) \cup \{0\} \quad a_1, a_2 : T \quad T \in \{\text{Int}, \text{BigInt}\} \\
 \llbracket \bar{a} \langle \Pi \wedge a_1 == a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \\
 \llbracket \bar{a} \langle \Pi \wedge a_1 > a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \quad \llbracket \bar{a} \langle \Pi \wedge a_1 < a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_3 \mid T_3 \rangle \\
 P' = a_1 == a_2 \wedge P_1 \vee a_1 > a_2 \wedge P_2 \vee a_1 < a_2 \wedge P_3 \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P' \mid \text{if } (a_1 == a_2) T_1 \text{ else if } (a_1 > a_2) T_2 \text{ else } T_3 \rangle
 \end{array}$$

GENERIC TYPE SPLIT – INPUT

$$\begin{array}{c}
 a_1 \in \bar{a} \quad a_2 \in \bar{a} \cup \text{bound}(\Pi) \quad a_1, a_2 : T \quad T \text{ is a type variable} \\
 \llbracket \bar{a} \setminus a_1 \langle \Pi[a_1 \mapsto a_2] \triangleright \phi[a_1 \mapsto a_2] \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \\
 \llbracket \bar{a} \langle \Pi \wedge a_1 \neq a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \quad P' = a_1 == a_2 \wedge P_1 \vee a_1 \neq a_2 \wedge P_2 \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P' \mid \text{if } (a_1 == a_2) \{T_1\} \text{ else } \{T_2\} \rangle
 \end{array}$$

GENERIC TYPE SPLIT – BOUND

$$\begin{array}{c}
 a_1 \in \text{bound}(\Pi) \quad a_2 \in \text{bound}(\Pi) \quad a_1, a_2 : T \quad T \text{ is a type variable} \\
 \llbracket \bar{a} \langle \Pi \wedge a_1 == a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \\
 \llbracket \bar{a} \langle \Pi \wedge a_1 \neq a_2 \triangleright \phi \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \quad P' = a_1 == a_2 \wedge P_1 \vee a_1 \neq a_2 \wedge P_2 \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P' \mid \text{if } (a_1 == a_2) \{T_1\} \text{ else } \{T_2\} \rangle
 \end{array}$$

INPUT SPLIT

$$\begin{array}{c}
 a \in \bar{a} \quad a : \text{Boolean} \quad \llbracket \bar{a} \setminus a \langle \Pi[a \mapsto \text{true}] \triangleright \phi[a \mapsto \text{true}] \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \\
 \llbracket \bar{a} \setminus a \langle \Pi[a \mapsto \text{false}] \triangleright \phi[a \mapsto \text{false}] \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle a \wedge P_1 \vee \neg a \wedge P_2 \mid \text{if } (a) \{T_1\} \text{ else } \{T_2\} \rangle
 \end{array}$$

BOUND VARIABLE SPLIT

$$\begin{array}{c}
 a \in \text{bound}(\Pi) \quad a : \text{Boolean} \\
 \llbracket \bar{a} \langle \Pi \wedge a \triangleright \phi \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \quad \llbracket \bar{a} \langle \Pi \wedge \neg a \triangleright \phi \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle a \wedge P_1 \vee \neg a \wedge P_2 \mid \text{if } (a) \{T_1\} \text{ else } \{T_2\} \rangle
 \end{array}$$

Figure 1.9 – Splitting synthesis rules

ADT SPLIT – INPUT

$$\begin{array}{l}
 a \in \bar{a} \quad a : T \quad T \text{ is an ADT with constructors } C_1(f_{11}, \dots, f_{1k_1}), \dots, C_n(f_{n1}, \dots, f_{nk_n}) \\
 \quad \quad \quad a_{ij}, i \in [1, n], j \in [1, k_i] \text{ are fresh variables} \\
 a_1 = C_1(a_{11}, \dots, a_{1k_1}) \quad \llbracket (\bar{a} \setminus a) \cup a_{11} \dots a_{1k_1} \langle \Pi[a \mapsto a_1] \triangleright \phi[a \mapsto a_1] \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \\
 \quad \quad \quad \dots \\
 a_n = C_n(a_{n1}, \dots, a_{nk_n}) \quad \llbracket (\bar{a} \setminus a) \cup a_{n1} \dots a_{nk_n} \langle \Pi[a \mapsto a_n] \triangleright \phi[a \mapsto a_n] \rangle x \rrbracket \vdash \langle P_n \mid T_n \rangle \\
 T = (a \text{ match } \{ \text{case } C_1(a_{11}, \dots, a_{1k_1}) \Rightarrow T_1 \dots \text{case } C_n(a_{n1}, \dots, a_{nk_n}) \Rightarrow T_n \}) \\
 P = (a \text{ match } \{ \text{case } C_1(a_{11}, \dots, a_{1k_1}) \Rightarrow P_1 \dots \text{case } C_n(a_{n1}, \dots, a_{nk_n}) \Rightarrow P_n \}) \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle
 \end{array}$$

ADT SPLIT – BOUND

$$\begin{array}{l}
 a \in \text{bound}(\Pi) \quad a : T \\
 T \text{ is an ADT with constructors } C_1(f_{11}, \dots, f_{1k_1}), \dots, C_n(f_{n1}, \dots, f_{nk_n}) \\
 \quad \quad \quad a_{ij}, i \in [1, n], j \in [1, k_i] \text{ are fresh variables} \\
 a_1 = C_1(a_{11}, \dots, a_{1k_1}) \\
 \llbracket \bar{a} \langle \Pi \wedge a.\text{isInstanceOf}[C_1] \wedge a_{11} \leftarrow a.f_{11} \wedge \dots \wedge a_{1k_1} \leftarrow a.f_{1k_1} \triangleright \phi[a \mapsto a_1] \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \\
 \quad \quad \quad \dots \\
 a_n = C_n(a_{n1}, \dots, a_{nk_n}) \\
 \llbracket \bar{a} \langle \Pi \wedge a.\text{isInstanceOf}[C_n] \wedge a_{n1} \leftarrow a.f_{n1} \wedge \dots \wedge a_{nk_n} \leftarrow a.f_{nk_n} \triangleright \phi[a \mapsto a_n] \rangle x \rrbracket \vdash \langle P_n \mid T_n \rangle \\
 T = (a \text{ match } \{ \text{case } C_1(a_{11}, \dots, a_{1k_1}) \Rightarrow T_1 \dots \text{case } C_n(a_{n1}, \dots, a_{nk_n}) \Rightarrow T_n \}) \\
 P = (a \text{ match } \{ \text{case } C_1(a_{11}, \dots, a_{1k_1}) \Rightarrow P_1 \dots \text{case } C_n(a_{n1}, \dots, a_{nk_n}) \Rightarrow P_n \}) \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle
 \end{array}$$

Figure 1.10 – ADT splitting synthesis rules

TERM EXPLORATION

$$\begin{array}{l}
 T = \text{EXPLORE}(\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket) \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle \text{true} \mid T \rangle
 \end{array}$$

Figure 1.11 – Sketch for term exploration rule

$$\begin{array}{lll}
 \text{argsSmaller}(i : \text{Int}, & i > 0 \wedge \Pi) & = \{i - 1\} \\
 \text{argsSmaller}(i : \text{Int}, & i < 0 \wedge \Pi) & = \{i + 1\} \\
 \text{argsSmaller}(i : \text{BigInt}, & i > 0 \wedge \Pi) & = \{i - 1\} \\
 \text{argsSmaller}(i : \text{BigInt}, & i < 0 \wedge \Pi) & = \{i + 1\} \\
 \text{argsSmaller}(C_T(f_1, \dots, f_n) : T, & \Pi) & = \bigcup \{ \{f_i\} \cup \text{argsSmaller}(f_i, \Pi) \mid f_i : T \} \\
 \text{argsSmaller}(v, & \Pi) & = \emptyset \quad \text{otherwise}
 \end{array}$$

Figure 1.12 – Computing smaller arguments for recursive calls

Recursive Calls. When synthesizing programs with recursive functions, a synthesis system needs to generate recursive calls to the function under synthesis, while guaranteeing that these calls will not introduce non-termination. In Leon, this is the responsibility of the rule `INTRODUCE RECURSIVE CALLS` defined in Figure 1.8. This rule binds a fresh variable *rec* to a recursive call to the function under synthesis, and appends the binding to the path condition. To make sure that no non-termination is introduced, the rule has to choose suitable arguments for this call. This is achieved by tracking sets of safe arguments in the path condition with a witness called *terminating* [KKK15, Kne16]. Terminating is written $\Downarrow[]$ and takes a function call as argument, for example, $\Downarrow[f(\bar{e})]$.

`INTRODUCE RECURSIVE CALLS` scans the path condition for clauses of the form $\Downarrow[f(\bar{e})]$, and generates recursive calls such that exactly one argument is *smaller* than the respective argument in \bar{e} , and all other arguments remain the same. When generating the initial synthesis problem, a terminating clause with the formal arguments of the function under synthesis is appended to the path condition. Notice that those arguments might be transformed by synthesis rules and thus will not always syntactically coincide with the formal arguments. Additionally, further terminating clauses might be added to the path condition by synthesis rules. In fact, `INTRODUCE RECURSIVE CALLS` appends to the path condition terminating clauses with the recursive calls it generates.

The “smaller argument” relation is interpreted as “closer to zero by 1” for integral types, and as transitive structural inclusion for ADTs. Although this relation is too restrictive to generate some specific programs, it coincides with patterns appearing often in functional programming. The set of smaller arguments of an expression is computed by the function *argsSmaller*, defined in Figure 1.12. In the definition, C_T indicates a type constructor of type T . *argsSmaller* will return the empty set unless the path condition passed to it (syntactically) indicates the existence of a smaller argument. Also, in the case of an ADT constructor, we only consider its fields f_i that are of the same type as the constructor itself. This is to make sure that the introduced call will be type-correct.

This approach cannot directly generate recursive calls where more than one argument changes; for example, it cannot generate a recursive call to a function that updates an accumulator while traversing a data structure. However, the rule adds the generated call to the terminating clauses of the path condition; this new call can be used in a next step to generate calls where an additional argument is smaller.

For instance, consider the example of Figure 1.13, which selects the first n elements of an input list. To solve this benchmark, Leon first instantiates `INEQUALITY SPLIT`, followed by `INTRODUCE RECURSIVE CALLS` and `ADT SPLIT`. After solving the resulting `Nil` subproblem, the “partial solution” of Figure 1.13 is reached. At this point, `INTRODUCE RECURSIVE CALLS` is instantiated again. The path condition contains the clause $\Downarrow[\text{take}[A](\text{Cons}[A](h, t), n - \text{BigInt}(1))]$. This is the call bound to *rec1* before, where the variable l has been transformed by `ADT SPLIT` into $\text{Cons}[A](h, t)$. Given that $\text{argssmaller}(\text{Cons}[A](h, t)) = \{t\}$, the rule generates the desired

```

def take[A](l: List[A], n: BigInt) : List[A] = {
  require(n ≥ 0)
  ???[List[A]]
} ensuring { (res: List[A]) ⇒
  ((l, n), res) passes {
    case (Nil(), _) ⇒ Nil()
    case (_, BigInt(0)) ⇒ Nil()
    case (Cons(a, Cons(b, Nil())), BigInt(1)) ⇒ Cons(a, Nil())
    case (Cons(a, Cons(b, Nil())), BigInt(2)) ⇒ Cons(a, Cons(b, Nil()))
    case (Cons(a, Cons(b, Nil())), BigInt(5)) ⇒ Cons(a, Cons(b, Nil()))
    case (Cons(a, Cons(b, Cons(c, Nil()))), BigInt(2)) ⇒ Cons(a, Cons(b, Nil())) } }

// Partial solution:
def take[A](l : List[A], n : BigInt): List[A] = {
  require(n ≥ BigInt(0))
  if (n == BigInt(0)) {
    List[A]()
  } else {
    val rec1 = take[A](l, n - BigInt(1))
    l match {
      case Nil() ⇒
        List[A]()
      case Cons(h, t) ⇒
        ???[List[A]] } } }

// Final solution
def take[A](l : List[A], n : BigInt): List[A] = {
  require(n ≥ BigInt(0))
  if (n == BigInt(0)) {
    List[A]()
  } else {
    l match {
      case Nil() ⇒
        List[A]()
      case Cons(h, t) ⇒
        val rec2 = take[A](t, n - BigInt(1))
        Cons[A](h, rec2) } } }

```

Figure 1.13 – Example of recursive call with two arguments changed

recursive call `take[A](t, n - BigInt(1))`. For the final solution, Leon simplifies away the useless call bound to `rec1`.

An alternative to having a separate deductive rule to generate recursive calls would be to generate them within term exploration, which is mentioned below. In fact, this approach was taken in previous versions of Leon, using a similar technique to discover smaller arguments. The benefit of our approach is that the variable bound to the result of the recursive call is available to subsequent decomposition rules. This enables new forms of programs to be synthesized. For example, consider the solution of the run-length example in Section 1.2.3: in the 6th line, the result of the recursive call is used to decide how to encode the head of the current list.

Other normalizing rules. `UNUSED INPUT` applies to a problem which includes an input parameter not present in either the path condition or specification of the problem. It generates a subproblem equivalent to the original problem, but without this input. In the rule definition, `FV` is a function which returns the free variables of an expression.

`ONE-POINT` immediately solves a problem whose specification is of the form $x == e$ with e as the solution.

`UNCONSTRAINED OUTPUT` solves a problem whose output is not constrained in the specification. This problem is solvable with any value of the output type, and we solve it with the “simplest value” of this type, computed with the `sv` function. Note that the simplest value might not exist, namely in the case where the type T is a type variable, or an ADT whose constructors all include a field of a generic type. In these cases, `UNCONSTRAINED OUTPUT` does not apply.

The `DETUPLE` rules break a composite input parameter or bound variable down to its components, i.e., its projections if it is a tuple, or its fields if it is a case class. When decomposing an input parameter, it is removed from the input parameter list and replaced with fresh parameters that correspond to its components. Appearances of the initial input parameter in the problem are also replaced. In contrary, a bound variable cannot be eliminated. Therefore, its components are bound to new variables and appended to the path condition instead.

Splitting rules. Splitting rules are decomposition rules that decompose the problem based on pattern matching or a boolean test.

Each of the two `INEQUALITY SPLIT` rules picks two terms of the same integral type chosen among (1) the constant 0, (2) the problem’s input parameters and (3) its bound variables. It introduces a comparison between these two terms and generates a subproblem for each branch of this comparison (including equality), adding the relevant clause to the path condition of each subproblem. The solution to the original problem is an three-part if-clause whose branches are the solutions of the respective subproblems.

In contrast to prior work [Kne16] on Leon, there are two versions of this rule: The first version (INPUT) is applied when at least one of the compared terms is an input variable. In this case, we can eliminate the input variable in the “equals” case by substituting it for the other term in the comparison. This is not possible in the case where none of the expressions is an input variable: in this case, we have to introduce an additional constraint in the path condition, similarly to the “greater” and “smaller” cases. This is shown in the BOUND version of the rule. In fact, each splitting rule of this section comes in two versions, similarly to INEQUALITY SPLIT.

An additional feature of splitting rules is that they try to avoid generating non-reachable subproblems: If one of the three branches can (syntactically) be shown to be unreachable by the problem’s path condition, the respective subproblem is not generated. This is omitted in the inference rules for readability purposes.

The GENERIC TYPE SPLIT rules are similar to INEQUALITY SPLIT, but they apply for inputs and bound variables of a generic type. Since a total order is not defined for generic types, these rules generate two branches instead, one for equality and one for inequality between the compared variables. Similarly, the INPUT SPLIT and BOUND VARIABLE SPLIT rules generate subproblems based on the value of a boolean input or bound variable.

The most complicated splitting rules are the ADT SPLIT rules, which generate subproblems by pattern matching on an input parameter or bound variable. Both versions generate one subproblem for each constructor of the type of the matched variable, and they compose the partial solutions with a pattern matching. The right-hand side of each case in this pattern matching is the term returned by the respective subproblem. The INPUT version of the rule introduces the binders of the pattern as new input parameters in place of the matched input. In contrary, the BOUND version cannot eliminate the matched bound variable. Therefore, it introduces the necessary information into the path condition of each subproblem: the precise type of the variable is asserted, and every field of the matched variable is bound to a fresh variable in the path condition.

Term Exploration. Eventually a problem is not able to be profitably decomposed further, and simple closing rules such as GROUND might not apply. Therefore, synthesis assumes the presence of a rule that is able to generate arbitrary terms in the target language and verify them against the specification. Such a rule is sketched in Figure 1.11: If a function EXPLORE – which we leave abstract for now – is able to find a term satisfying the specification, then the rule succeeds with this term. If EXPLORE finds no such term, the rule fails. The approach we follow for EXPLORE in this dissertation is based on enumerating programs derived from a context-free grammar. In Chapter 2 we explain how we construct suitable grammars, and in Chapter 3 we present the two variants of term exploration used in Leon.

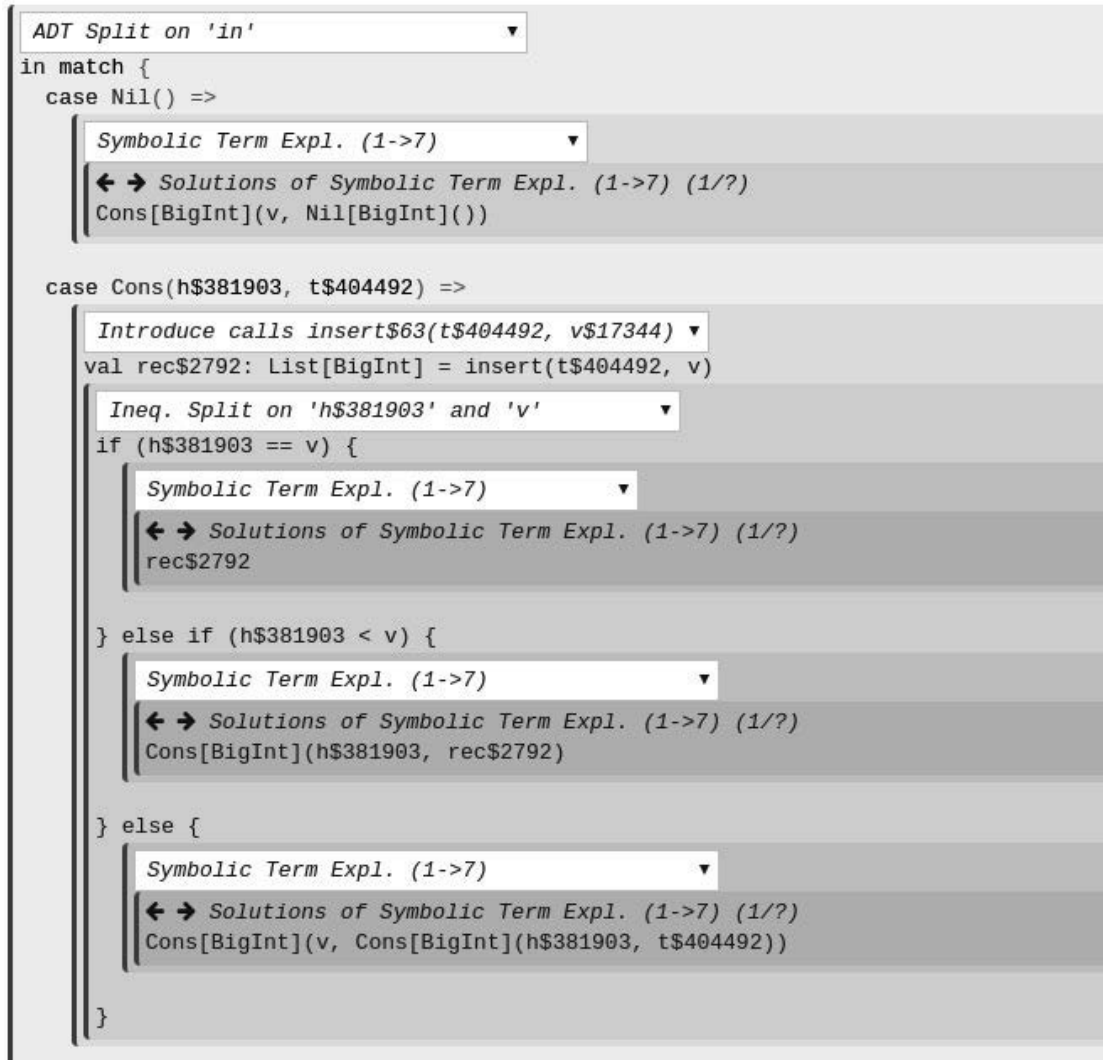


Figure 1.14 – Synthesis graph for sorted list insertion

1.5.7 Example of a Synthesis Graph

Figure 1.14 presents the synthesis graph corresponding to the list insertion problem of Section 1.2.2. The image is a screen capture from Leon’s online interface [Leo]. Every framed square represents a synthesis problem, or an AND-node in the graph. Rule instantiations that do not constitute part of the final solution are not shown, thus the OR-nodes of the synthesis graph are not explicit in the image.

1.6 Conclusion

In this chapter, we presented an overview of deductive synthesis in Leon. After providing a few motivating examples, we formalized the synthesis problem, including a special notion

of path conditions. We then presented the input language handled by Leon: we presented its purely functional component with different synthesis constructs, we implemented an object-oriented extension, and we introduced the syntax for symbolic examples. We then explained how deductive synthesis rules are used to break down or immediately solve a synthesis problem. Compared to previous work in synthesis, we presented a new approach to handle recursive calls, where the result of the recursive call is bound to a variable in the path condition of the problem and can be used in further synthesis rules. Also, we presented a set of updates to the deductive synthesis rules that are able to handle path conditions with bound variables. We showed that these advances allow us to synthesize problems that are not synthesizable with the previous versions of Leon.

2 Term Grammars

In Section 1.5.6 we mentioned *term exploration*, a deductive rule capable of discovering arbitrary programs that satisfy a given specification from a space of candidate programs. The space of discoverable programs is expressed in Leon by a *term grammar*. A term grammar is a context-free grammar (CFG) or a probabilistic context-free grammar (PCFG), whose non-terminal symbols describe a set of constraints on generated expressions. Such constraints include at minimum the type of generated expressions, but can also include additional information. Essentially, a term grammar defines a model of the target language. This model is sensitive to the context of the currently synthesized term: for instance, it takes into account visible variables, functions and user-defined types. A parse tree derived from the grammar corresponds to a program term.

Generating and manipulating term grammars is an integral part of our system, as the choice of grammar can dramatically influence the size of the program space and the efficiency of discovering interesting programs therein, thus affecting the effectiveness of synthesis.

Grammars with types as nonterminals, such as those defined in Section 2.1, have been used previously in the synthesis community to represent program spaces [ABJ⁺13, PGGP14], as well as in previous work on Leon. The content of Section 2.2 was first introduced in a previous publication by the author and others [KKK16], but is formalized and thoroughly analyzed here. Sections 2.3 to 2.7 were discussed in a previous publication by the author and others [KRKK17].

2.1 Definition

Definition 2.1. A term grammar is a triple $G = (\mathcal{N}, \mathcal{R}, \mathcal{S})$, where:

- (1) \mathcal{N} is a finite, non-empty set of nonterminal symbols, where each nonterminal $N \in \mathcal{N}$ is associated with a type $T_N \in \mathcal{T}$, where \mathcal{T} is the finite set of language types.
- (2) \mathcal{R} maps each nonterminal N to a finite set of production rules $\mathcal{R}(N)$. Each production

rule $R \in \mathcal{R}(N)$ is a well-typed construct of the form $f(N_1, N_2, \dots, N_k)$, where f is the top-level operator and N_1, N_2, \dots, N_k are the child nonterminal symbols, such that the output types of f and N coincide: $T_f = T_N$. The rule $R = f(N_1, N_2, \dots, N_k)$ is also assigned the type T_f . It might be the case that $N_i = N_j$ for $i \neq j$, but every appearance of a child nonterminal is treated as a separate object, i.e., every index i appears exactly once.

(3) $\mathcal{S} \in \mathcal{N}$ is the starting symbol of the grammar.

Note that \mathcal{N} does not necessarily coincide with the set of types of the target language; we only require that the mapping from nonterminals to types be surjective. For grammars where the mapping is bijective, or if we want to signify that a particular nonterminal symbol represents all possible values of its corresponding type, we will allow a nonterminal symbol to be represented by its corresponding type, and write T_N instead of N . We call such grammars *plain*.

Definition 2.2. A plain grammar is a term grammar where $\mathcal{N} = \mathcal{T}$, where \mathcal{T} is the set of types of the target language.

We represent term grammars in common grammar notation: the m rules for nonterminal N will be written

$$\begin{array}{l} N ::= \mathbf{f}(N_{11}, N_{12}, \dots, N_{1k_1}) \\ \quad \dots \\ \quad | \quad \mathbf{f}(N_{m1}, N_{m2}, \dots, N_{mk_m}) \end{array}$$

We use *italic* script for nonterminal symbols, and **bold** script for terminal symbols of the grammar. We will use the same scripts to represent abstract entities that belong to the respective categories. For instance, in the above, $\mathbf{f}(N_{11}, N_{12}, \dots, N_{1k_1})$ represents the application of an arbitrary operator \mathbf{f} of the target language on unknown nonterminal symbols N_{1i} . We will call \mathbf{f} a *terminal operator* if it has 0 operands, and a *nonterminal operator* if it has more than 0 operands.

Generally, a CFG represents a set of words, or sequences of terminals, of the target language. In our setting, it is more useful to think of a term grammar as representing a set of *parse trees*, which correspond to abstract syntax trees (ASTs) of the target language. We can assume that every production rule is implicitly enclosed in parentheses; this way, parse trees correspond one-to-one to words, so we do not have to distinguish between them.

To produce a parse tree from a grammar, we perform a *derivation*: Starting from a node labeled with the starting symbol as our current parse tree, we perform a series of *expansions* or *applications* of production rules until no nonterminal symbol remains in the tree. Expanding a production rule means choosing a nonterminal node N in the parse tree and a rule $R = f(N_1, N_2, \dots, N_k) \in \mathcal{R}(N)$, and substituting this leaf node N in the parse tree with a node that contains f as label and N_1, N_2, \dots, N_k as children. We write \mathcal{E}_N for the set of all derivations of a nonterminal N . We will discuss derivations more in Section 3.3.

$$\begin{array}{c}
e ::= 0 \mid 1 \mid x \mid e + e \mid \text{if } (e) \{e\} \text{ else } \{e\} \mid e \leq e \mid e \&\& e \\
\\
\frac{}{0, 1, x : \text{Int}} \quad \frac{e_1 : \text{Int} \quad e_2 : \text{Int}}{e_1 + e_2 : \text{Int}} \quad \frac{e_1 : \text{Boolean} \quad e_2 : \text{Int} \quad e_3 : \text{Int}}{\text{if } (e_1) \{e_2\} \text{ else } \{e_3\} : \text{Int}} \\
\\
\frac{e_1 : \text{Int} \quad e_2 : \text{Int}}{e_1 \leq e_2 : \text{Boolean}} \quad \frac{e_1 : \text{Boolean} \quad e_2 : \text{Boolean}}{e_1 \&\& e_2 : \text{Boolean}}
\end{array}$$

Figure 2.1 – A simple expression language and its type system

As an example, consider a grammar generating terms for the expression language of Figure 2.1, whose type system is also listed in the same figure. x is an available integer variable.

$$\begin{array}{lcl}
Int & ::= & 0 \\
& & \mid 1 \\
& & \mid x \\
& & \mid Int + Int \\
& & \mid \text{if } (Boolean) \{Int\} \text{ else } \{Int\} \\
Boolean & ::= & Int \leq Int \\
& & \mid Boolean \&\& Boolean
\end{array}$$

As explained above, this is a plain term grammar, since the nonterminals *Int* and *Boolean* correspond one-to-one to the types of the language and carry no additional information. Note the well-typedness of the above grammar according to the type system of the language: The productions of the nonterminal *Int* expand to expressions of type *Int*, and the productions of *Boolean* expand to expressions of type *Boolean*. Also, all operators are applied to operands of the expected types.

When we use grammars for synthesis, the type of the starting symbol coincides with the type of the term we want to synthesize, i.e., the type of the output variable of the synthesis problem: $T_{\mathcal{G}} = T_x$.

Describing generated terms of a specific category just with their type often does not capture all information that we want the synthesizer to consider. This results in the generation of many undesired terms, which in turn slows down the process of synthesis. Also, there is no distinction between different terms as to which are more desirable than others. To address these problems, in the next sections we describe other, more sophisticated variants of term grammars.

2.2 Aspect Grammars

The main shortcoming of plain term grammars is that they generate too many redundant terms. For example, the grammar of Section 2.1 would generate both equivalent terms $(1 + 1) + x$ and $1 + (1 + x)$, as well as all three of x , $x + 0$ and $x + 0 + 0$.

One approach to address this problem would be to develop grammars which incorporate knowledge about the underlying language, e.g., properties of operators. For example, we could incorporate into the grammar the knowledge that a 0 should not be generated as an operand of $+$. Given the constants 0 and 1, the operator $+$, and an available variable x , such a grammar could look as follows:

$$\begin{aligned} Int &::= 0 \mid NZ \\ NZ &::= 1 \mid x \mid NZ + NZ \end{aligned}$$

The problem with this approach is its lack of modularity: the grammar would have to be developed in a monolithic way. This means that a small change in the design of the grammar, e.g., adding an operator to the language or coming up with an additional optimization, would require significant effort to implement, since all grammar optimizations would have to be taken into account. This is especially problematic for synthesis, where the grammar has to depend on variables, functions and user-defined types in the context of the synthesis problem.

In this section, we introduce a more effective solution to this problem. The idea is to develop a plain grammar (where nonterminals coincide with term types) separately first, and then attach to its nonterminals additional information we call *aspects*. We will write $T_{\{A\}}$ to represent a nonterminal consisting of type T enhanced with aspect A . We can attach multiple aspects on the same type T : $T_{\{A_1\}...\{A_n\}}$.

To compute the production rules for $T_{\{A\}}$, we apply on the set of rules for T a *transformer* $\llbracket A \rrbracket$ that is specific to A . The final set of rules for a nonterminal $T_{\{A_1\}...\{A_n\}}$ is computed by successive application of the transformers of all attached aspects A_i on the initial set of rules for T .

Aspects can be viewed as serving two functionalities:

- Imposing normal forms on generated terms to avoid generation of redundant terms.
- Implementing other constraints we wish to impose in specific contexts. For example, specific implementations of term exploration require that we fix the size of generated terms, as discussed in Section 3.2. Additionally, in Section 4.5.1 we use aspects in the context of repair to generate a set of terms containing small modifications to an given term.

Aspects appeared in previous work by the author of this dissertation and others [KKK16] as *attributes*. In this dissertation, we renamed them to *aspects* to avoid confusion with attribute

grammars as defined by Knuth [Knu68].

In [PGGP14], the authors apply a similar disambiguation technique. In their case, the disambiguation occurs after the terms have been generated with syntactic checks on the generated terms. In contrast, our aspects affect the grammar itself, meaning that all terms produced are automatically good candidates. Another disambiguation technique called *indistinguishability modulo inputs* [URD⁺13] approximates program equivalence with equivalence on a fixed set of inputs. This technique merges more equivalent expressions than aspect grammars but has the overhead of evaluation. We explore this technique as part of our probabilistic enumeration algorithm in Section 3.3.

2.2.1 Example

Consider the grammar from Section 2.1:

$$\begin{aligned} \text{Int} &::= 0 \mid 1 \mid x \\ &\mid \text{Int} + \text{Int} \\ &\mid \text{if } (\text{Boolean}) \{ \text{Int} \} \text{ else } \{ \text{Int} \} \\ \text{Boolean} &::= \text{Int} \leq \text{Int} \\ &\mid \text{Boolean} \&\& \text{Boolean} \end{aligned}$$

If a user requests terms for Int , they will receive, among others, the obviously undesired terms $x + 0$, $0 + 0$ etc. Thankfully, our system provides an aspect n that eliminates operations with neutral elements of operands. Knowing that, the user can set the starting symbol of the grammar to $\text{Int}_{\{n\}}$ instead. When this happens, the grammar will compute terms for $\text{Int}_{\{n\}}$ by applying $\llbracket n \rrbracket$ on the rules for Int . The result is the following:

$$\begin{aligned} \text{Int}_{\{n\}} &::= 0 \\ &\mid 1 \\ &\mid x \\ &\mid \text{Int}_{\{n\}\{\neg 0\}} + \text{Int}_{\{n\}\{\neg 0\}} \\ &\mid \text{if } (\text{Boolean}_{\{n\}}) \{ \text{Int}_{\{n\}} \} \text{ else } \{ \text{Int}_{\{n\}} \} \end{aligned}$$

Observe that the new rules contain the newly encountered nonterminals $\text{Int}_{\{n\}\{\neg 0\}}$ and $\text{Boolean}_{\{n\}}$. The grammar must now generate rules for these symbols. $\neg 0$ is an aspect that filters out rules whose right-hand side is the constant 0. Like before, to generate rules for $\text{Int}_{\{n\}\{\neg 0\}}$, the grammar will start from rules for $\text{Int}_{\{n\}}$ and apply $\llbracket \neg 0 \rrbracket$ on them.

The rest of the grammar then looks as follows:

$$\begin{aligned}
 Int_{\{n\}\{\neg 0\}} &::= 1 \\
 &| \times \\
 &| Int_{\{n\}\{\neg 0\}} + Int_{\{n\}\{\neg 0\}} \\
 &| \text{if } (Boolean_{\{n\}}) \{Int_{\{n\}}\} \text{ else } \{Int_{\{n\}}\} \\
 Boolean_{\{n\}} &::= Int_{\{n\}} \leq Int_{\{n\}} \\
 &| Boolean_{\{n\}\{\neg c\}} \&\& Boolean_{\{n\}\{\neg c\}} \\
 Boolean_{\{n\}\{\neg c\}} &::= Int_{\{n\}} \leq Int_{\{n\}} \\
 &| Boolean_{\{n\}\{\neg c\}} \&\& Boolean_{\{n\}\{\neg c\}}
 \end{aligned}$$

The aspect $(\neg c)$ signifies that boolean constants are disallowed, though it has no effect on this particular grammar.

The aspect grammar above will not generate any additions with 0 as operand, and our initial goal is achieved.

2.2.2 Formalization

An aspect grammar $G = (\mathcal{N}, \mathcal{R}, \mathcal{S})$ is a context-free grammar defined in terms of a plain grammar $G_0 = (\mathcal{N}_0, \mathcal{R}_0, \mathcal{S}_0)$ and a *finite* set of aspects \mathcal{A} . Since G_0 is a plain grammar, its nonterminals coincide with the (finite) set of language types \mathcal{T} : $\mathcal{N}_0 = \mathcal{T}$. Each aspect $A \in \mathcal{A}$ is accompanied by a *transformer* $\llbracket A \rrbracket$, which maps each pair (N, \bar{R}) of a nonterminal and a set of rules to another set of rules \bar{R}' . We require that the sets \bar{R} and \bar{R}' are finite, and that all rules they contain are of the same type as N .

Definition 2.3. *Given a plain grammar $G_0 = (\mathcal{N}_0, \mathcal{R}_0, \mathcal{S}_0)$ and a set of aspects \mathcal{A} , an aspect grammar $G = (\mathcal{N}, \mathcal{R}, \mathcal{S})$ is defined as follows:*

- \mathcal{N} is the set of types $\mathcal{T} = \mathcal{N}_0$ annotated with any number of aspects in \mathcal{A} .

$$\mathcal{N} = \{ T_{\{A_1\} \dots \{A_n\}} \mid T \in \mathcal{T}, n \geq 0, A_i \in \mathcal{A}, A_i \neq A_j \text{ for } i \neq j \}.$$

- The definition of $\mathcal{R}(N)$ depends on whether N is annotated with aspects or not:

$$\begin{aligned}
 \mathcal{R}(T) &= \mathcal{R}_0(T) \\
 \mathcal{R}(T_{\{A_1\} \dots \{A_n\}}) &= \llbracket A_n \rrbracket(N, \mathcal{R}(N)), \quad \text{where } N = T_{\{A_1\} \dots \{A_{n-1}\}}.
 \end{aligned}$$

- \mathcal{S} is \mathcal{S}_0 annotated with an arbitrary number of aspects, i.e., it can be chosen arbitrarily from the set

$$\mathcal{S} \in \{ \mathcal{S}_{0\{A_1\} \dots \{A_n\}} \mid n \geq 0, A_i \in \mathcal{A}, A_i \neq A_j \text{ for } i \neq j \}.$$

Note that in practice we do not have to compute the production rules for every possible

nonterminal, but only those reachable from the starting symbol.

Theorem 2.1. *Aspect grammars are context-free.*

Proof. Based on the definition of \mathcal{N} and the finiteness conditions for types, aspects and aspect transformers, \mathcal{N} is finite.

From the definition of \mathcal{R} , its elements have the form of rules of context-free grammars. Also, \mathcal{R} is finite according to the finiteness condition for aspect transformers.

Finally, the starting symbol $\mathcal{S} \in \mathcal{N}$. □

2.2.3 Definitions of Aspects

In this section, we describe several of the aspects that we define in Leon.

Neutral and Absorbing Elements

Several arithmetic and boolean operators have so-called neutral or absorbing operands that are sources of redundancies. For example, terms such as $e + 0$, $e / 1$, or $e * 0$ are all equivalent to a shorter form. We eliminate these from the grammar by using an aspect n that excludes neutral elements from operands. The use of this aspect was demonstrated by an example in Section 2.2.1.

Formally,

$$\llbracket n \rrbracket(N, (R_1, \dots, R_n)) = (R'_1, \dots, R'_n),$$

where, if

$$R_i = \mathbf{f}(N_1, N_2, \dots, N_k),$$

we have

$$R'_i = \mathbf{f}(N'_1, N'_2, \dots, N'_k),$$

with

$$N'_i = N_{i\{n\}\{\neg C\}},$$

where C is the set of all neutral and absorbing elements of f for its i -th argument.

If C is a set of operators (possibly including nullary operators such as constants), the aspect $\neg C$ is defined as follows:

$$\llbracket \neg C \rrbracket(N, \bar{R}) = \{ \mathbf{f}(N_1, N_2, \dots, N_k) \mid \mathbf{f}(N_1, N_2, \dots, N_k) \in \bar{R} \wedge \mathbf{f} \notin C \}.$$

Note that, apart from removing appropriate operands, n also propagates itself to the nonterminals in the right-hand side of the rule.

Associative Operators

To remove redundancy caused by operator associativity, we require that all associative operators associate to the left. To impose this property, we introduce an aspect *ao*.

Formally,

$$\llbracket ao \rrbracket(N, (R_1, \dots, R_n)) = (R'_1, \dots, R'_n),$$

where, if

$$R_i = f(N_1, N_2, \dots, N_k),$$

we have

$$R'_i = f(N'_1, N'_2, \dots, N'_k),$$

with

$$N'_i = \begin{cases} N_{i\{ao\}\{\neg f\}}, & \text{if } f \text{ is an associative binary operator and } i = 2, \\ N_{i\{ao\}}, & \text{otherwise.} \end{cases}$$

Similar to *n*, the aspect *ao* will, along with its main functionality, propagate itself to the nonterminals in the right-hand side of the rule.

Ground Terms

Our grammars should not generate ground terms for two reasons: first, because different combinations of ground terms can end up simplifying to equivalent programs (consider $1 + 3$ and $2 + 2$) and second, because ground terms can be discovered much more efficiently by the **GROUND** rule, that invokes only a satisfiability query on the Leon solver instead of trying to solve the general synthesis problem of Equation 1.1 with nested quantifiers.

Let the aspect *G* denote that a ground term is expected, whereas $\neg G$ that a non-ground term is expected.

Formally, $\llbracket G \rrbracket(N, \tilde{R}) = \tilde{R}_1 \cup \tilde{R}_2$ where

$$\begin{aligned} \tilde{R}_1 &= \{ f(N_{1\{G\}}, N_{2\{G\}}, \dots, N_{n\{G\}}) \mid f(N_1, N_2, \dots, N_n) \in \tilde{R}, n > 0 \} \\ \tilde{R}_2 &= \{ f() \mid f() \in \tilde{R}, f \text{ is a ground term} \} \end{aligned}$$

In other words, *G* maintains nonterminal operators with ground operands, and terminal operators which are ground terms.

Also, $\llbracket \neg G \rrbracket(N, \tilde{R}) = \tilde{R}_1 \cup \tilde{R}_2$ where

$$\begin{aligned} \tilde{R}_1 &= \{ f(N_{1\{G_1\}}, N_{2\{G_2\}}, \dots, N_{n\{G_n\}}) \mid f(N_1, N_2, \dots, N_n) \in \tilde{R}, n > 0, G_i \in \{G, \neg G\}, \neg G \in \{G_i\} \} \\ \tilde{R}_2 &= \{ f() \mid f() \in \tilde{R}, f \text{ is not a ground term} \} \end{aligned}$$

Intuitively, for nonterminal operators, $\neg G$ imposes that at least one operand is non-ground, and terminal operators are maintained only if they are non-ground.

When invoking term exploration with aspect grammars in synthesis, we attach $\neg G$ to the starting symbol of the grammar to avoid generating ground terms.

Size and Commutative Operators

In specific variants of term exploration, we want to constrain the size of generated terms. Having assigned an arbitrary cost to every production rule in the grammar, the *size* or *cost* of an expression is defined as the total cost of all rules expanded in its derivation.

To generate terms of a specific size, we introduce an aspect named *Sized*. A similar mechanism was used in previous work [Kne16]. *Sized* is a parametric aspect, taking the desired size of generated expression as a parameter. The size has to be a constant natural number and, since only nonterminals of smaller sizes can be generated from a given sized nonterminal, the condition of the finiteness of aspects is maintained. We will denote *Sized* with size s with norm notation $|s|$. For instance, $Int_{|5|}$ produces only integer expressions of size 5, such as $a + b + c$ if all operators have size 1.

Formally,

$$\llbracket |s| \rrbracket (N, \bar{R}) = \bar{R}',$$

where for each $f (N_1, N_2, \dots, N_n) \in \bar{R}$, we distinguish two cases:

- if $n > 0$, i.e. f is a nonterminal operator, then

$$f (N_{1\{|s_1|\}}, N_{2\{|s_2|\}}, \dots, N_{n\{|s_n|\}}) \in \bar{R}'$$

for all combinations of $s_i > 0$ such that $size(f) + \sum s_i = s$.

Additionally, if f is a commutative operator, we only include a rule in R' if $\forall i < j. s_i \geq s_j$. As a result, only left-heavy terms are produced by the grammar (i.e., $(a * b) + c$ and not the equivalent $c + (a * b)$). This does not completely eliminate redundancies introduced by commutative operators, but doing so would require more expressive grammars, or to inspect and reject terms after they have been generated.

- If $n = 0$, i.e., f is a terminal operator, then $f() \in R'$ if and only if $size(f) = s$.

2.2.4 Comparison to other Grammar Formalisms

As shown in Theorem 2.1, aspect grammars are context-free given the finiteness conditions of Section 2.2.2. However, for the sake of completeness, we will compare aspect grammars with other grammar formalisms that extend context-free grammars.

Attribute grammars were suggested by Knuth [Knu68] as a way to enhance context-free grammars with context-sensitive information called *attributes*. Attributes are attached to the nonterminals (syntactic categories) of the grammar and can be elements of any domain, such as integers, booleans or ADTs. Attributes are divided into two categories: *synthesized* and *inherited* attributes. The synthesized attributes of a nonterminal N are computed in terms of the attributes of the child nonterminals in an expanded rule for N . Inherited attributes for a nonterminal N are computed in terms of the attributes of the parent and sibling nonterminals of N in an expanded rule of which N is a child nonterminal. Additionally, *attribute conditions* can be defined, which are predicates on the values of the attributes. The derivation of a sentence is valid if and only if the context-free part of the grammar is satisfied and all attribute conditions are true. Observe that attributes are examined during parsing, as opposed to our aspects, which are only used to compute production rules during a preprocessing stage of the grammar. Nevertheless, our aspects can be associated with inherited attributes, as they are computed top-down. In contrast to inherited attributes, our aspects are taken from a finite set, and are not constrained to filtering out rules, but can implement arbitrary transformations between rules.

Two-level grammars [vWMP⁺77, CU77] is another extension of context-free grammars. Instead of a finite set, the set of nonterminals of a two-level grammar is defined as the language recognized by a second context-free grammar, called the *metagrammar*. Since the set of nonterminals of a two-level grammar is potentially infinite, two-level grammars are very expressive; in fact, they have been proven to be Turing-complete. This raises the question if we would have a similar effect if we relaxed the finiteness condition for nonterminals for aspect grammars.

Theorem 2.2. *Aspect grammars without the finiteness condition for nonterminals are Turing-complete.*

Proof. Let G_u be an unrestricted grammar over alphabet Σ , with nonterminals \mathcal{N}_u , starting nonterminal \mathcal{S}_u , and let \rightarrow_{G_u} be the expansion relation for G_u . We will define an aspect grammar G which recognizes the same language as G_u .

Following Section 2.2.2, define G_0 as follows: $\mathcal{N}_0 = \{N_0\}$, $\mathcal{R}_0 = \emptyset$, $\mathcal{S}_0 = N_0$, for some nonterminal N_0 . Define $\mathcal{A} = \{s \mid s \in (\Sigma \cup \mathcal{N}_u)^*\}$, i.e., \mathcal{A} coincides with the set of strings over $\Sigma \cup \mathcal{N}_u$. Then for every $s \in \mathcal{A}$, define

$$\llbracket s \rrbracket(N, \bar{R}) = \begin{cases} \{s\}, & \text{if } s \in \Sigma^* \\ \{N_{0\{s'\}} \mid s \rightarrow_{G_u} s'\}, & \text{otherwise} \end{cases} \quad (2.1)$$

Note that $\llbracket s \rrbracket$ is a constant function of N and \bar{R} for any given s . Finally, define $\mathcal{S} = N_{0\{\mathcal{S}_u\}}$.

From the second line of Equation 2.1, it is easy to show by induction on \rightarrow_{G_u} that $N_{0\{s\}}$ is derived by G if and only if s is derived by G_u . Also, from the first line of Equation 2.1, $s \in \Sigma^*$

is derived by G if and only if $N_{0\{s\}}$ is derived by G . Therefore, G_u and G define the same language. \square

In Leon, we chose to impose the finiteness conditions of Section 2.2.2. This way, the resulting grammars remain context-free, which facilitates the algorithms we developed for their exploration. At the same time, we found that they are expressive enough for the purposes of this dissertation.

2.2.5 Implementation

In practice, the n , ao and G aspects of Section 2.2.3 are all defined in tandem. Every production rule in the built-in grammar of Leon, and optionally in user-defined grammars (Section 2.6), is tagged with a tag that indicates what kind of operator it represents. Every child nonterminal is passed its parent's tag, its position in its parent's operands, as well as a ternary value corresponding to G , $\neg G$, or the absence of either. Those values together constitute a composite aspect incorporating the functionality of all three aspects, and is used to compute the production rules for the annotated nonterminal.

A quantitative analysis on the effect of aspects on synthesis is presented in Section 3.4.

2.3 Probabilistic Term Grammars

Although aspect grammars capture some knowledge about a functional language, they implement an “all or nothing” strategy: in a given context, every rule is either applicable or not, with no quantitative differentiation among applicable rules. A more accurate modelling of the target language would also require quantifying how likely it is for an expression to be useful in a given context. To that end, in this section we introduce the use of *probabilistic term grammars* to our synthesis system.

Definition 2.4. A probabilistic term grammar is a triple $G = (\mathcal{N}, \mathcal{R}, \mathcal{P})$ as in Definition 2.1, with the additional attribute that

- (4) each rule R is associated with a probability $p_R \in [0, 1)$ such that for all nonterminals N ,
- $$\sum_{R \in \mathcal{R}(N)} p_R = 1.$$

We will represent a probabilistic grammar similar to other grammars, indicating the probability of each rule on its right-hand side:

$$\begin{array}{ll} N ::= & \mathbf{f}(N_{11}, N_{12}, \dots, N_{1k_1}) \quad (p = p_1) \\ & \dots \\ & | \quad \mathbf{f}(N_{m1}, N_{m2}, \dots, N_{mk_m}) \quad (p = p_m) \end{array}$$

We require the probability of each rule to be $p_R < 1$ (note the strict inequality). If we relax this assumption, there are some technical difficulties in setting up the probability space, and results such as Theorem 3.2 will need additional constraints to be true.

Given a derivation $e \in \mathcal{E}_N$, we define its probability, $P(e)$, as the product of the probabilities of all rules used in e . We assume that $\sum_{e \in \mathcal{E}_N} P(e) = 1$, i.e., $P(e)$ assign true probabilities to the parse trees generated by the grammar. This property of probabilistic grammars is called *consistency* [BT73]. In our context, the consistency of a grammar is not important: As we will see in Section 3.3, where we discuss a probabilistic term exploration algorithm for synthesis, we use probabilities only as a total order among parse trees, which is not affected by the sum of those probabilities. Additionally, considering that any practical synthesis algorithm will explore a finite number of parse trees, we could conceptually normalize the probabilities of discovered trees by their sum, thus obtaining a consistent grammar without affecting the total order they define.

2.4 Generic Term Grammars

Generic term grammars offer a concise way of representing operations on polymorphic types. For this version of term grammars, types associated with nonterminals may be polymorphic, and production rules may be parametrized by type parameters.

Definition 2.5. A generic (probabilistic) term grammar is defined according to Definition 2.1 (resp. 2.4), except clauses (1) and (2) are modified as follows:

- (1) Types associated with nonterminals may be polymorphic.
- (2) Each production rule $R = t(N_1, N_2, \dots, N_k) \in \mathcal{R}(N)$ may additionally be parameterized by a sequence of type parameters \bar{A} . Those have to be a subset of the type parameters of the types T_N and T_{N_1}, \dots, T_{N_k} of the nonterminals in the rule. Rules have to be well typed, as in Definition 2.1 (resp. 2.4). We will denote a rule parametrized by \bar{A} as $\forall \bar{A}. N ::= t(N_1, N_2, \dots, N_k)$.

For instance, we could describe common operations on a generic list data-structure with the following grammar:

```

 $\forall A. \text{List}[A] ::= \text{Cons}(A, \text{List}[A])$ 
 $\forall A. \text{List}[A] ::= \text{Nil}[A]()$ 
 $\forall A. A ::= \text{List}[A].\text{head}$ 
 $\forall A. \text{BigInt} ::= \text{List}[A].\text{size}$ 
 $\text{BigInt} ::= 0$ 

```

When looking for productions of a specific ground type, we instantiate generic productions that *can* return that type and introduce them as normal monomorphic productions. For

example, if we are looking for productions of $List[BigInt]$, we consider the first two rules above instantiated with $A \mapsto BigInt$, and the third instantiated with $A \mapsto List[BigInt]$.

More complications arise when type parameters are only bound in the right-hand side of a rule, such as the fourth rule above. Such type parameters can be instantiated with arbitrary types, which, in the presence of parametric types, can generate an infinite number of ground productions. For example, the rule “ $\forall A. BigInt ::= List[A].size$ ” can be instantiated with $A = \{BigInt, List[BigInt], List[List[BigInt]], \dots\}$. This is not compatible with the definition of term grammars, that requires types to be finite.

To solve this conflict, we abandon the theoretical completeness of the instantiation by considering only a set of *reasonable* types \mathcal{T}_r during instantiation. This set of types is initialized with all types returned by ground productions, then iteratively expanded by discovering new return types of \mathcal{T}_r -instantiations of generic productions. For example, starting with $\mathcal{T}_r = \{BigInt\}$, running one iteration yields $\mathcal{T}_r = \{BigInt, List[BigInt]\}$. By keeping the number of such iterations finite, we ensure that the set \mathcal{T}_r also remains finite.

The number of discovery iterations we perform has a strong practical impact on the number of final productions, hence on performance. On benchmarks with several generic data structures, this discovery is typically exponential. Therefore, we need a heuristic to bound the size of discovered types. The heuristic we found to be working reasonably as a bound for type T is an underapproximation of the size of the smallest expression of type T .

In the case of a probabilistic grammar, we normalize the probabilities of a nonterminal after we instantiate generic productions for it.

2.5 Built-In Grammar

The built-in grammar of Leon is a union of

- primitive operations, including arithmetic and boolean operators and constants
- operations on sets, tuples and maps
- all constants in the body of the function under synthesis
- case class constructors for ADTs defined in the program
- input and bound variables of the synthesis problem
- function calls on functions in the same module as the function under synthesis
- (optionally) recursive calls to the function under synthesis. We give the user the option to include recursive calls in the grammar, as opposed to using the dedicated synthesis rule that introduces them.

The built-in grammar is not probabilistic. For term enumeration algorithms that require probabilities, those are set to be uniform for the rules of each nonterminal.

2.5.1 Value Grammar

Leon also defines a grammar that generates only PureScala values (ground terms). This grammar is useful, for instance, when we want to enumerate a large number of test inputs for a given function, as is the case in Section 4.3.1. The value grammar consists of

- Small literals for the `BigInt` and `Int` types, as well as the **true**, **false** and unit literals
- Small tuple literals and small sets containing values
- Case class constructors for ADTs defined in the program whose fields are values.

2.6 Custom Grammars

The built-in grammars of Leon offer a good generic solution for any synthesis problem. However, this may not be sufficient for all applications. Firstly, the grammars are not probabilistic, and it would be difficult to hard-code meaningful probabilities for all types. Secondly, depending on the application, a user may want to provide their own specialized grammar to model the programs they target more accurately. Therefore, we give the user the possibility to define their own grammars. We present a Scala file format in which a user can define custom probabilistic grammars. Additionally, as those files can be very tedious to author manually, we provide a simple method to automatically extract a grammar file from a corpus of code.

2.6.1 Grammar Files

A grammar file represents a probabilistic term grammar in a specially formatted Scala file, where each production rule is represented as a Scala function. Since grammars are not used directly but go through some phases of preprocessing (instantiation of generic rules and application of aspects) which would invalidate probabilities associated with the user-defined rules, we defer from using probabilities directly in the grammar files. Instead, the user associates an integer measure frequency to each function. The generated probability for each rule is proportional to its associated measure.

Plain nonterminals. In the simplest case, nonterminals coincide with Scala types. To explain this file format, let us look at an example grammar file describing a set of simple arbitrary-precision integer programs:

```
@production(10) def plus(a: BigInt, b: BigInt): BigInt = a + b
@production(5)  def minus(a: BigInt, b: BigInt): BigInt = a - b
```

```

@production(10) def o: BigInt = BigInt(1)
@production(5) def z: BigInt = BigInt(0)
@production(20) def vBigInt: BigInt = variable[BInt]

```

A function annotated with `@production` is treated as a grammar production rule. A rule of the form $T ::= f(T_1, T_2, \dots, T_k)$, is represented in a grammar file as

```
def prodF(a1: T1, ..., ak: Tk): T = f(a1,...,ak).
```

The function's name is arbitrary. The annotation's argument indicates the integer measure associated to the rule. The relative frequencies will be computed from the absolute frequencies during grammar preprocessing.

Note that f above can be any Scala expression that belongs to the fragment parsable by Leon according to Section 1.4, as long as this expression refers to each function parameter exactly once. This is consistent with Definition 2.1. This way, rules can represent arbitrarily complicated Scala functions.

An invocation of the built-in function `variable[T]`, for some type T , indicates the absolute frequency of generating any variable of type T . This built-in compensates for the fact that we do not know a priori the names of the variables available to each specific synthesis problem. During preprocessing, this production will be instantiated to a concrete production for each available variable of the correct type, with the probability of the variable rule distributed equally among those variables.

For example, when synthesizing a function with two parameters (x : `BigInt`, y : `BigInt`), the above grammar file would generate the following grammar:

```

BigInt ::= BigInt + BigInt  ( $p_+ = 0.2$ )
         | BigInt - BigInt  ( $p_- = 0.1$ )
         | 0                ( $p_0 = 0.2$ )
         | 1                ( $p_1 = 0.1$ )
         | x                ( $p_x = 0.2$ )
         | y                ( $p_y = 0.2$ )

```

Grammars with aspects. It is possible attach aspects to custom grammar rules. Rather than giving the aspects directly, the user annotates the rules with the set of tags used by the default grammar to compute aspects, as mentioned in Section 2.2.5. The system will then compute the aspects based on those tags and operand positions as explained in Section 2.2.5. The tags are attached to rules with the `@tag` Scala annotation. For example, `@tag("commut")` indicates that the tagged function corresponds to a rule with a top-level commutative operator, which is taken into account, for example, by the Sized aspect of Section 2.2.3.

Annotated nonterminals. In Section 2.1 we left the definition of grammar nonterminals open as long as they are mappable to types. With this in mind, custom grammars offer the possibility to annotate nonterminals with additional custom information. This is useful if a user wants to express restrictions on generated expressions that are not expressible by the provided aspects and do not want to develop a new aspect.

An annotated nonterminal is represented in a grammar file with an implicit class. Implicit classes in Scala always take a single (constructor) parameter and automatically define a coercion from the type of their argument to their own type. For example, the program

```
implicit class C(i: Int) { def foo = i + 1 }
println(42.foo)
```

will print 43 to the standard output. We utilize this functionality to skip repeating the definition of the nonterminal in every rule and make the grammar file more readable. The type of the constructor argument is the type associated with the nonterminal in the term grammar. Annotated nonterminals are annotated in grammar files with the @label Scala annotation.

For example, suppose a user wants to manually specify that the grammar of Section 2.6.1 should not generate 0 operands for +, as well as in the second position of -. This can be handled by aspects, but can also be captured with the following annotated grammar:

<i>BigInt</i>	::=	<i>BigInt-Toplevel</i>	(<i>p</i> = 1.0)
<i>BigInt-Toplevel</i>	::=	<i>BigInt-Nonzero</i>	(<i>p</i> = 0.8)
		0	(<i>p</i> = 0.2)
<i>BigInt-Nonzero</i>	::=	<i>BigInt-Nonzero</i> + <i>BigInt-Nonzero</i>	(<i>p</i> ₁ = 0.25)
		<i>BigInt-Toplevel</i> - <i>BigInt-Nonzero</i>	(<i>p</i> ₁ = 0.125)
		x	(<i>p</i> _x = 0.25)
		y	(<i>p</i> _y = 0.25)
		1	(<i>p</i> ₁ = 0.125)

All nonterminals of the above grammar have type *BigInt*.

This grammar is represented by the following grammar file:

```
1 @label implicit class Nonzero(val v: BigInt)
2 @label implicit class Toplevel(val v: BigInt)
3
4 @production(1) def start(b: Toplevel): BigInt = b.v
5
6 @production(40) def nz2Bi(nz: Nonzero): Toplevel = nz.v
7 @production(10) def z: Toplevel = BigInt(0)
8
9 @production(10) def plus(a: Nonzero, b: Nonzero): Nonzero = a.v + b.v
```

```
10 @production(5) def minus(a: Toplevel, b: Nonzero): Nonzero = a.v - b.v
11 @production(20) def vNZ: Nonzero = variable[BigInt]
12 @production(5) def oNZ: Nonzero = BigInt(1)
```

Observe how the implicit notation makes the grammar file less cluttered: For example, in line 6 of the above code, the term `nz.v` of type `BigInt` is coerced to `Toplevel`.

2.7 Extracting Grammars from Corpus

Whereas manually specifying grammars is realistic and useful for DSLs and small languages, it would be tedious to expect a user-provided grammar for generic programming tasks, where numerous types and operators have to be handled. To address this, we provide an automated system to extract a grammar from a corpus of Scala programs. We provide two different grammar extractors, described in the following paragraphs.

2.7.1 Extractor for Plain Grammars

The plain grammar extractor counts the occurrences of each operator in the corpus and outputs a function with the corresponding frequency for each one. Literals of different values, as well as functions of different names, count as different operators. In contrary, variable names are not preserved, and all variables of a specific type are summarized in a single rule invoking the variable built-in function. Generic applications of the operator (for instance, in the case of equality) generate generic productions. Generated productions are tagged to aid the application of aspects, as explained in the previous section.

For example, a corpus containing only the expression $((x * y) * 2)$, where $x, y: \text{BigInt}$, would result in the following grammar file:

```
@production(1) @tag("const")
def pBigIntInfiniteIntegerLiteral0(): BigInt = BigInt(2)
@production(2) @tag("times")
def pBigIntTimes(v0 : BigInt, v1 : BigInt): BigInt = v0 * v1
@production(2) @tag("top")
def pBigIntVariable(): BigInt = variable[BigInt]
```

The `"top"` tag is the default tag, which has no effect on aspect transformers.

Given available input parameters $a, b, c: \text{BigInt}$, The above file will generate the following grammar:

<i>BigInt</i>	<code>::=</code>	<code>2</code>	$(p = 0.2)$
	<code> </code>	<code>BigInt * BigInt</code>	$(p = 0.4)$
	<code> </code>	<code>a</code>	$(p = 0.133)$
	<code> </code>	<code>b</code>	$(p = 0.133)$
	<code> </code>	<code>c</code>	$(p = 0.133)$

2.7.2 Extractor for Annotated Grammars

The second extractor uses annotated grammars to extract conditional operator frequencies. Specifically, the frequency of an operator is conditional its operand position and its parent node in the abstract syntax tree. The assumption is that these grammars will work better because they encode more information about the structure of programs in the target language. For example, these grammars can encode that a 0 will never be used as an operand of a + operator.

As an example, a corpus containing only the expression $((x * y) * 2)$ as before would produce the following grammar file:

```
@label implicit class BigInt_TOPLEVEL(val v : BigInt)
@label implicit class BigInt_0_Times(val v : BigInt)
@label implicit class BigInt_1_Times(val v : BigInt)

@production(1)
def pBigIntStart(v0 : BigInt_TOPLEVEL): BigInt = v0.v

@production(1)
def pBigIntTimes(v0 : BigInt_0_Times, v1 : BigInt_1_Times): BigInt_TOPLEVEL =
  v0.v * v1.v

@production(1)
def pBigIntVariable(): BigInt_0_Times = variable[BigInt]

@production(1)
def pBigIntVariable(): BigInt_1_Times = variable[BigInt]

@production(1)
def pBigIntInfiniteIntegerLiteral(): BigInt_1_Times = BigInt(2)
```

Given an input parameters a, b: BigInt, this file corresponds to the following grammar:

$BigInt$	$::=$	$BigInt_{\{Top\}}$	$(p = 1.0)$
$BigInt_{\{Top\}}$	$::=$	$BigInt_{\{*,0\}} * BigInt_{\{*,1\}}$	$(p = 1.0)$
$BigInt_{\{*,0\}}$	$ $	a	$(p_a = 0.5)$
	$ $	b	$(p_b = 0.5)$
$BigInt_{\{*,1\}}$	$::=$	a	$(p_a = 0.25)$
	$ $	b	$(p_b = 0.25)$
	$ $	2	$(p_2 = 0.5)$

2.7.3 Extraction of Grammars from a Corpus of Leon Benchmarks

We ran the two grammar extractors on a corpus of programs consisting of 172 Leon verification benchmarks, totalling a number of 15715 lines of code. This is a small corpus, but due to the language restrictions of PureScala, at present our system cannot analyze publicly available Scala code.

In total, we extracted 59165 expressions of 18 types. From these expressions, we filtered out expressions of user-defined types except library types such as List, as those are not relevant for unrelated synthesis problems. Additionally, we filtered out **match**-expressions, since those are introduced by rules such as ADT SPLIT and are not meant to be generated by enumeration rules. After this filtering, we end up with 21128 expressions. Those were then analyzed separately by the plain and annotated grammar extractors. The plain grammar extractor output a total of 258 rules, while the annotated output 1118 rules and 247 nonterminals (@label **implicit** classes).

We evaluate the two generated grammars for synthesis in Section 3.4.

2.8 Extended Example

As a case study about how grammars are generated and processed by Leon, we look in more depth at an example taken from synthesis of the run-length encoding benchmark of Section 1.2.3. Specifically, we pick a problem generated after the application of a few synthesis rules, when Leon is attempting to synthesize the recursive branch of the function where the first two elements of the list are equal.

The partial solution at this point looks as follows:

```
def encode[A](l : List[A]): List[(BigInt, A)] = {
  | match {
    | case Nil() =>
      | Nil()
    | case Cons(h, t) =>
      | encode[A](t) match {
```

```

case Nil() =>
  List((BigInt(1), h))
case Cons(h1 @ (h_1, h_2), t1) =>
  if (h == h_2) {
    ???[List[(BigInt, A)]] // Current synthesis problem
  } else {
    ???[List[(BigInt, A)]]
  } } } }

```

The position of the current problem is indicated with a comment in the code.

Figure 2.2 shows the built-in grammar for this problem before instantiating generic rules and applying aspects. Ground productions are followed by generic productions. Since the built-in grammar is pretty large and includes productions for types not relevant for this problem, such as sets and maps, we have omitted productions corresponding to irrelevant types for presentation purposes. In practice, those additional productions do not negatively influence the performance of the system, since only nonterminals that are reachable from the starting nonterminal are considered by our enumeration algorithms.

Note that there is an entry for lists of type A , the type parameter of the input list l . This rule is not parametric to A , as A refers to a type parameter found in the program and thus is treated as a type constant (Skolem constant).

A term enumeration algorithm called Symbolic Term Exploration (STE, Section 3.2) uses this grammar to explore solutions for the problem. STE needs to explore terms by increasing size, therefore it uses the Sized aspect. It also applies the composite aspect of Section 2.2.5, which implements the functionality of the aspects for ground terms, neutral/absorbing elements, and associative operators presented in Section 2.2.3. Eventually, it discovers a solution to the problem of size 7.

Figure 2.3 shows a part of the resulting grammar after instantiation of generic rules and application of the aspects. As a reminder, aspects are applied only on demand, depending on which aspects the user attached to the starting symbol of the grammar. The first part of the grammar corresponds to the starting symbol of type $List[(BigInt, A)]$. The first aspect attached to it is the sized aspect for size=7. The second aspect is the composite aspect indicating the generated expression is a top-level expression (without a parent), at position 0, and there is no indication about its groundness. The first rule for this symbol corresponds to an instantiation of the decode generic rule, and the rest are instantiations of the Cons generic rule for all applicable of sizes and ground statuses. The *FunCall* and *Cns* tags indicate that the parent of the current expression is respectively a function call and an ADT constructor. Rules corresponding to $t1$ and the generic rule for Nil did not generate any rules when transformed by the Sized aspect for this size. Next, some rules are listed corresponding to integer expressions of size 1, namely those that are with + operators. We include a nonterminal with the empty list


```

BigInt ::= 0
          | 1
          | BigInt + BigInt
          | BigInt - BigInt
          | BigInt * BigInt
          | h_1
Boolean ::= false
            | true
            | ! Boolean
            | Boolean && Boolean
            | Boolean || Boolean
            | BigInt < BigInt
            | BigInt ≤ BigInt
List[A] ::= t
            | A ::= h
List[(BigInt, A)] ::= t1
∀ T      Boolean ::= T == T
∀ T      | legal[T]( List[(BigInt, T)] )
∀ T1, T2 (T1, T2) ::= (T1, T2)
∀ T      List[T] ::= decode[T]( List[(BigInt, T)] )
∀ T      | Cons[T]( T , List[T] )
∀ T      | Nil[T]()

```

Figure 2.2 – Built-in grammar before processing

of rules ϵ to indicate it is non-productive. Finally, the single rule for expressions of type *A* and size 1 is shown.

Figure 2.4 lists the plain grammar of Section 2.7.3 for the same problem. The frequencies depicted on the right are absolute frequencies. Generic rules have already been instantiated. The tuple rule and its instantiation is not extracted from the corpus and has been manually inserted. The reason the rule **t** for *List*[*A*] is duplicated is that the custom grammar contains @production functions for variables of both generic types *T* and *List*[*T*]. Such clauses do not make it into the grammar, because they are not grammar rules per say but rule templates (they need a specific variable to become proper rules). Figure 2.5 shows part of the same grammar after calculation of probabilities and instantiation of aspects. The negative logarithmic probability for each rule is displayed.

Finally, Figure 2.6 shows part of the annotated extracted grammar after processing. Recall that the difference with the previous grammar is that there is no aspect application, and the probabilities listed are directly based on frequencies extracted from the corpus.

$$\begin{aligned}
 List[(BigInt, A)]_{\{7\}\{Top, 0\}} &::= \text{decode}[(BigInt, A)](List[(BigInt, (BigInt, A))]_{\{6\}\{FunCall, 0, \neg G\}} \\
 &| \text{Cons}[(BigInt, A)]((BigInt, A)_{\{1\}\{Cns, 0, \neg G\}} , \\
 &List[(BigInt, A)]_{\{5\}\{Cns, 1, \neg G\}}) \\
 &| \text{Cons}[(BigInt, A)]((BigInt, A)_{\{1\}\{Cns, 0, G\}} , \\
 &List[(BigInt, A)]_{\{5\}\{Cns, 1, \neg G\}}) \\
 &| \text{Cons}[(BigInt, A)]((BigInt, A)_{\{1\}\{Cns, 0, \neg G\}} , \\
 &List[(BigInt, A)]_{\{5\}\{Cns, 1, G\}}) \\
 &\dots \\
 &| \text{Cons}[(BigInt, A)]((BigInt, A)_{\{5\}\{Cns, 0, \neg G\}} , \\
 &List[(BigInt, A)]_{\{1\}\{Cns, 1, \neg G\}}) \\
 &| \text{Cons}[(BigInt, A)]((BigInt, A)_{\{5\}\{Cns, 0, G\}} , \\
 &List[(BigInt, A)]_{\{1\}\{Cns, 1, \neg G\}}) \\
 &| \text{Cons}[(BigInt, A)]((BigInt, A)_{\{5\}\{Cns, 0, \neg G\}} , \\
 &List[(BigInt, A)]_{\{1\}\{Cns, 1, G\}}) \\
 &\dots \\
 BigInt_{\{1\}\{+, 0, \neg G\}} &::= h_1 \\
 BigInt_{\{1\}\{+, 0, G\}} &::= \epsilon \\
 BigInt_{\{1\}\{+, 1, \neg G\}} &::= h_1 \\
 BigInt_{\{1\}\{+, 1, G\}} &::= \text{BigInt}(1) \\
 A_{\{1\}\{Cns, 1, \neg G\}} &::= h
 \end{aligned}$$

Figure 2.3 – Part of grammar of figure 2.2 after processing

$(\text{BigInt}, A) ::= (\text{BigInt}, A)$	$f = 1$
$A ::= \mathbf{h}$	$f = 618$
$\text{BigInt} ::= - \text{BigInt}$	$f = 28$
$\quad \text{BigInt} / \text{BigInt}$	$f = 34$
$\quad \text{BigInt} - \text{BigInt}$	$f = 281$
$\quad \text{BigInt} \bmod \text{BigInt}$	$f = 2$
$\quad \text{BigInt} + \text{BigInt}$	$f = 214$
$\quad \text{BigInt} \% \text{BigInt}$	$f = 19$
$\quad \text{BigInt} * \text{BigInt}$	$f = 143$
$\quad \mathbf{h_1}$	$f = 3289$
$\quad \text{BigInt}(0)$	$f = 562$
$\quad \text{BigInt}(1)$	$f = 375$
$\quad \text{BigInt}(-1)$	$f = 11$
$\quad \text{BigInt}(2)$	$f = 97$
$\text{Boolean} ::= \text{Boolean} \ \&\& \ \text{Boolean}$	$f = 1784$
$\quad \mathbf{true}$	$f = 476$
$\quad \mathbf{false}$	$f = 221$
$\quad \text{Boolean} == \text{Boolean}$	$f = 82$
$\quad \text{List}[\text{BigInt}] == \text{List}[\text{BigInt}]$	$f = 18$
$\quad \text{BigInt} == \text{BigInt}$	$f = 350$
$\quad \text{Boolean} \rightarrow \text{Boolean}$	$f = 50$
$\quad \text{BigInt} \leq \text{BigInt}$	$f = 349$
$\quad \text{BigInt} < \text{BigInt}$	$f = 242$
$\quad \mathbf{!} \text{Boolean}$	$f = 486$
$\quad \text{Boolean} \ \ \text{Boolean}$	$f = 139$
$\text{List}[(\text{BigInt}, A)] ::= \mathbf{Cons}[(\text{BigInt}, A)] \ ((\text{BigInt}, A), \text{List}[(\text{BigInt}, A)])$	$f = 62$
$\quad \mathbf{t1}$	$f = 1498$
$\quad \mathbf{Nil}[(\text{BigInt}, A)]()$	$f = 80$
$\text{List}[A] ::= \mathbf{t}$	$f = 618$
$\quad \mathbf{t}$	$f = 880$
$\text{List}[\text{BigInt}] ::= \mathbf{Cons}[\text{BigInt}] \ (\text{BigInt}, \text{List}[\text{BigInt}])$	$f = 40$
$\quad \mathbf{Nil}[\text{BigInt}]()$	$f = 31$
$\forall T \quad \text{Boolean} ::= T == T$	$f = 22$
$\forall T \quad \quad \text{List}[T] == \text{List}[T]$	$f = 42$
$\forall T_1, T_2 \quad (T1, T2) ::= (T1, T2)$	$f = 1$
$\forall T \quad \text{List}[T] ::= \mathbf{Nil}[T]()$	$f = 80$
$\forall T \quad \text{List}[T] ::= \mathbf{Cons}[T](T, \text{List}[T])$	$f = 62$

Figure 2.4 – Plain custom grammar before preprocessing

$List[(BigInt, A)]_{\{Top, 0\}} ::=$	$Cons[(BigInt, A)] \ ((BigInt, A)_{\{Cns, 0, \neg G\}} ,$	
	$\quad List[(BigInt, A)]_{\{Cns, 1, \neg G\}})$	$-\log(p) = 3.35$
	$ Cons[(BigInt, A)] \ ((BigInt, A)_{\{Cns, 0, G\}} ,$	
	$\quad List[(BigInt, A)]_{\{Cns, 1, \neg G\}})$	$-\log(p) = 3.35$
	$ Cons[(BigInt, A)] \ ((BigInt, A)_{\{Cns, 0, \neg G\}} ,$	
	$\quad List[(BigInt, A)]_{\{Cns, 1, G\}})$	$-\log(p) = 3.35$
	$ t1$	$-\log(p) = 0.17$
	$ Nil[(BigInt, A)]()$	$-\log(p) = 3.10$
	\dots	
$BigInt_{\{+, 0, \neg G\}} ::=$	$-BigInt_{\{-, 0, \neg G\}}$	$-\log(p) = 5.13$
	$ BigInt_{\{*, 0, \neg G\}} * BigInt_{\{*, 1, \neg G\}}$	$-\log(p) = 3.59$
	$ BigInt_{\{*, 0, G\}} * BigInt_{\{*, 1, \neg G\}}$	$-\log(p) = 3.59$
	$ BigInt_{\{*, 0, \neg G\}} * BigInt_{\{*, 1, G\}}$	$-\log(p) = 3.59$
	$ BigInt_{\{\%, 0, \neg G\}} \% BigInt_{\{\%, 1, \neg G\}}$	$-\log(p) = 5.52$
	$ BigInt_{\{\%, 0, G\}} \% BigInt_{\{\%, 1, \neg G\}}$	$-\log(p) = 5.52$
	$ BigInt_{\{\%, 0, \neg G\}} \% BigInt_{\{\%, 1, G\}}$	$-\log(p) = 5.52$
	$ BigInt_{\{mod, 0, \neg G\}} \bmod BigInt_{\{mod, 1, \neg G\}}$	$-\log(p) = 7.77$
	$ BigInt_{\{mod, 0, G\}} \bmod BigInt_{\{mod, 1, \neg G\}}$	$-\log(p) = 7.77$
	$ BigInt_{\{mod, 0, \neg G\}} \bmod BigInt_{\{mod, 1, G\}}$	$-\log(p) = 7.77$
	$ BigInt_{\{-, 0, \neg G\}} - BigInt_{\{-, 1, \neg G\}}$	$-\log(p) = 2.83$
	$ BigInt_{\{-, 0, G\}} - BigInt_{\{-, 1, \neg G\}}$	$-\log(p) = 2.83$
	$ BigInt_{\{-, 0, \neg G\}} - BigInt_{\{0, 1, G\}}$	$-\log(p) = 2.83$
	$ BigInt_{\{/, 0, \neg G\}} * BigInt_{\{/, 1, \neg G\}}$	$-\log(p) = 4.94$
	$ BigInt_{\{/, 0, G\}} * BigInt_{\{/, 1, \neg G\}}$	$-\log(p) = 4.94$
	$ BigInt_{\{/, 0, \neg G\}} * BigInt_{\{/, 1, G\}}$	$-\log(p) = 4.94$
	$ h_1$	$-\log(p) = 0.37$
$BigInt_{\{+, 0, G\}} ::=$	-1	$-\log(p) = 2.33$
	$ 2$	$-\log(p) = 0.10$

Figure 2.5 – Part of plain custom grammar after preprocessing

$List[(BigInt, A)]_{\{Top, 0\}} ::= \mathbf{Cons}[(BigInt, A)] \left((BigInt, A)_{\{Cns, 0\}}, \right.$	
$\left. List[(BigInt, A)]_{\{Cns, 1\}} \right)$	$-\log(p) = 3.28$
$\quad \mathbf{t1}$	$-\log(p) = 0.09$
$\quad \mathbf{Nil}[(BigInt, A)]()$	$-\log(p) = 3.02$
$List[(BigInt, A)]_{\{Cns, 1\}} ::= \mathbf{Cons}[(BigInt, A)] \left((BigInt, A)_{\{Cns, 0\}}, \right.$	
$\left. List[(BigInt, A)]_{\{Cns, 1, \neg G\}} \right)$	$-\log(p) = 3.66$
$\quad \mathbf{t1}$	$-\log(p) = 0.57$
$\quad \mathbf{Nil}[(BigInt, A)]()$	$-\log(p) = 0.89$
\dots	
$BigInt_{\{+, 0\}} ::= BigInt_{\{-, 0\}} - BigInt_{\{-, 1\}}$	$-\log(p) = 2.70$
$\quad BigInt_{\{*, 0\}} * BigInt_{\{*, 1\}}$	$-\log(p) = 2.82$
$\quad BigInt_{\{+, 0\}} + BigInt_{\{+, 1\}}$	$-\log(p) = 2.82$
$\quad \mathbf{h_1}$	$-\log(p) = 0.29$
$\quad \mathbf{BigIng(1)}$	$-\log(p) = 2.82$
$BigInt_{\{+, 1\}} ::= BigInt_{\{*, 0\}} * BigInt_{\{*, 1\}}$	$-\log(p) = 2.79$
$\quad \mathbf{h_1}$	$-\log(p) = 0.93$
$\quad \mathbf{BigIng(1)}$	$-\log(p) = 0.64$
$\quad \mathbf{BigIng(2)}$	$-\log(p) = 5.02$
$\quad \mathbf{BigIng(3)}$	$-\log(p) = 5.02$

Figure 2.6 – Part of annotated custom grammar after preprocessing

2.9 Conclusion

In this chapter, we presented an overview of different variants of term grammars that are used in Leon to describe program spaces. Those variants include probabilistic and generic grammars, as well as aspect grammars, a new technique to modularly describe the basic structure of a grammar and prior knowledge on the underlying language's operators. We showed the built-in grammar of Leon, in contrast with user-defined grammars, described in a Scala file with special annotations. We provide a system which can extract such a grammar from a corpus of code, and use it on a small corpus of Leon verification benchmarks.

3 Term Exploration

As mentioned in Chapter 1, Leon synthesis relies on a deductive synthesis rule capable of discovering a term from a space of candidate terms and verifying its correctness. In this chapter, we present two instantiations of such rule. The first one, called *Symbolic Term Exploration* [KKKS13, Kne16], operates on term grammars without probabilities, and explores terms in order of increasing size. The second one, which we call *Probabilistic Top-Down Term Enumeration* or just Probabilistic Term Enumeration (PTE), operates on probabilistic term grammars and explores terms in order of decreasing probability. Both rules are based on the counterexample-guided inductive synthesis (CEGIS) framework, which we briefly present next.

Parts of this chapter appeared in previous work by the author of this dissertation and others [KKK16, KRKK17].

3.1 Counter-Example Guided Inductive Synthesis

Counter-example guided inductive synthesis (CEGIS) is a widely used framework to describe program synthesis algorithms [SLTB⁺06, GJTV11]. Recall that the problem of synthesis is summarized by Formula (1.1): $\exists T. \forall \bar{a}. [\Pi \rightarrow \phi[x \mapsto T]]$. The main technical difficulty with finding a T satisfying this formula is the universally quantified “ $\forall \bar{a}$ ”, where the input variables \bar{a} can range over a large, or even potentially infinite domain. CEGIS circumvents this difficulty by approximating the universal quantification for all inputs with finite quantification over a set of known inputs that grows as the exploration advances.

We present CEGIS in Algorithm 3.1. At each step, the algorithm maintains a finite set A of concrete input examples. It is parameterized by two sub-procedures, Search and Verify.

In the Search phase, the algorithm finds an expression T that satisfies the specification for the

Chapter 3. Term Exploration

Algorithm 3.1: $\text{CEGIS}(\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket, A)$.

Repeat forever:

- (1) Let $T = \text{Search}(\bar{a}, A, \Pi, \phi, x)$.
 - (2) Let $\text{res} = \text{Verify}(\bar{a}, \Pi, \phi, x, T)$.
 - (3) If $\text{res} = \text{VALID}$, then return T .
 - (4) Otherwise, if $\text{res} = \text{INVALID}(\bar{a}_{\text{cex}})$, update $A := A \cup \{\bar{a}_{\text{cex}}\}$.
-

current set of concrete inputs \bar{a} or, formally,

$$\bigwedge_{\bar{a} \in A} \Pi \rightarrow \phi[x \mapsto T]. \quad (3.1)$$

Observe that, for a specific choice of $\bar{a} \in A$, determining whether a given T satisfies requirement (3.1) reduces to simply evaluating the resulting expression. Often, the strategy for implementing `Search` is to enumerate all candidate expressions, in some order, until a suitable answer is found.

In `Verify`, we check whether the candidate expression T works for all inputs \bar{a} by sending the formula $\forall \bar{a}. \Pi \rightarrow \phi[x \mapsto T]$ to the Leon verifier. If it does, we have found a satisfactory solution and the algorithm terminates. If not, `Verify` will provide a new input point for which the candidate program fails, which will be appended to A and will be used to further refine the results of `Search` in the next iteration of the algorithm.

One point that has to be clarified is how `Verify` handles recursive calls to the function under synthesis. `Verify` uses a version of the function containing all known information about the currently constructed synthesis solution. Recall that a term exploration rule is a closing rule of the Leon deductive framework, i.e., a leaf in the synthesis search graph (see Section 1.5). At any moment, the current graph describes the structure of the current (potentially incomplete) synthesis solution. `Verify` uses this graph to reconstruct the current solution and uses it in recursive calls.

As an example, suppose we are synthesizing the list insertion function of Section 1.2.2. For reference, its synthesis graph is depicted in Figure 1.14. Suppose that, as we are trying to solve the second-to-last subproblem (the $h < v$ case), we are verifying the term `rec` as a potential solution. `rec` is the variable bound to the introduced recursive call. At this moment, the incomplete synthesis solution looks as follows:

```
def insert(in : List[BigInt], v : BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Nil() => List(v)
```



```

case Cons(h, t) ⇒
  val rec = insert(t, v)
  if (h == v) rec
  else if (h < v) rec // Current subproblem
  else ???[List[BigInt]]
} ensuring {
  (out : List[BigInt]) ⇒
    out.content == in.content ++ Set[BigInt](v) && isSorted(out) }

```

This version of the function contains all known information about the current (incomplete) solution, and will be used in recursive calls to the function. In this example, although the term `rec` itself contains no obvious recursive calls, the formula sent to the verifier ($(\forall h, t, v. \Pi \rightarrow \phi[out \mapsto rec])$) does, since Π contains the binding ($rec \mapsto insert(t, v)$).

In the following sections, we describe two implementations of term exploration based on CEGIS, focusing on the Search procedure. The first one explores terms generated from a grammar in order of increasing size, and incorporates techniques for improved use of concrete examples. The second one is one of the few synthesis procedures which incorporate probabilities and probabilistic search into the CEGIS framework.

3.2 Symbolic Term Exploration

Symbolic Term Exploration (STE) is an adaptation of the existing algorithm of Leon and is presented in Algorithm 3.2. It is an instantiation of CEGIS. For clarity and performance reasons, STE does not separate the SEARCH and VERIFY steps of Algorithm 3.1.

Algorithm 3.2 takes as input a synthesis problem, a grammar G and a maximum size max of generated terms. During the initialization phase of the algorithm, a priority queue I is populated with input or input-output examples that are being tracked in the synthesizer, and some additional examples that are generated on the fly. All priorities in the queue are initialized to 0; they will be increased later as examples are used to filter out invalid programs. Another variable P is initialized to the empty set of programs.

UNFOLD produces all derivations of the grammar G for a specific size s . This is done with the help of the Sized aspect of Section 2.2.3: the starting symbol of the grammar is annotated with $|s|$, and all derivations of G are enumerated. Observe that a grammar annotated with Sized is never recursive, hence the generated set of programs is finite. This set of programs P is first filtered with the current set of available tests I . This is implemented with function CONCRETETEST in Algorithm 3.3, which we will explain later.

Every program p that passes concrete execution for all tests is sent to the verifier. Similarly to VERIFY in the previous section, SETRECURSIVECALLS(p) makes sure recursive calls to the function under synthesis use the current incomplete synthesis solution, including the current

Chapter 3. Term Exploration

Algorithm 3.2: $\text{STE}(\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x \rrbracket, G, \text{max})$

```
var  $I \leftarrow \text{QUEUE}(\text{initial examples})$ 
var  $P \leftarrow \emptyset$ 
for  $s \leftarrow 1$  to  $\text{max}$  do
   $P \leftarrow \text{UNFOLD}(G, s)$ 
   $P \leftarrow \text{CONCRETETEST}(P, I, I, \phi)$ 
  forall  $p \in P$  do
     $\text{SETRECURSIVECALLS}(p)$ 
     $f \leftarrow (\Pi \wedge \neg \phi[x \mapsto p])$ 
     $s \leftarrow \text{LEONSMT}(f)$ 
    switch  $s$  do
      case  $\text{UNSAT}$  do
         $\text{return } p$ 
      case  $\text{SAT}(a_0)$  do
         $P \leftarrow P \setminus \{p\}$ 
         $I \leftarrow \text{ENQUEUE}(I, a_0)$ 
         $P \leftarrow \text{CONCRETETEST}(P, \text{QUEUE}(a_0), I, \phi)$ 
return  $\text{FAIL}$ 
```

Algorithm 3.3: $\text{CONCRETETEST}(P, Q, I, \phi)$

```
 $P' \leftarrow \emptyset$ 
forall  $p \in P$  do
   $\text{SETRECURSIVECALLS}(p)$ 
  forall  $i \in Q$ ; // by order of priority in Q
  do
    switch  $i$  do
      case  $\bar{a}_0$  do
         $e \leftarrow \phi[x \mapsto p, \bar{a} \mapsto \bar{a}_0]$ 
      case  $(\bar{a}_0, x_0)$  do
         $e \leftarrow "p[\bar{a} \mapsto \bar{a}_0] == x_0"$ 
      if  $\neg \text{EXECUTE}(e)$  then
         $\text{INCREASEPRIORITY}(I, i)$ 
        break forall
   $P' \leftarrow P' \cup \{p\}$ 
return  $P'$ 
```

term p . Then, the Leon verifier is invoked with the negation of the specification, where the output variable has been substituted with the current term p . If the result is UNSAT, the program is valid and is returned; otherwise, the solver returns a new counterexample which is added to the set of examples I . The current program p is rejected and the new example is immediately used to potentially reject more programs in P , by invoking CONCRETETEST an additional time with a new queue containing only the new counterexample a_0 .

This process is repeated until the maximum term size is exceeded, at which point the algorithm fails.

Let us now take a closer look at CONCRETETEST. It takes as input the set of programs to be tested P , the priority queue of examples to be used Q , the main priority queue of examples in STE I , and the problem specification ϕ . I is passed so we can modify the priorities of its elements, as we will see later. First, a set of programs P' to be returned is initialized to the empty set. Every program is tested against every example with concrete execution: if the example i is an input example, we check if the specification is satisfied; if it is an input-output example, we check that the program evaluates to the output value of the example. The latter case corresponds to the expression in quotes. The examples are used in order of their priority in Q . The EXECUTE function will return true if and only if its argument evaluates to **true** without failing any specification, including those invoked in other functions or recursive calls on the function under synthesis. If p passes all examples, we add it to P' to be returned. In case p fails for an example i , we do not execute it on the remaining tests. Also, we increase the priority of i in the queue I : since it was useful to reject a program, we want it to be used earlier in the testing process. We found this optimization to be quite useful in practice.

3.2.1 Sizes of Operators

To apply the Sized aspect, we need to assign a natural number as size to every production rule of our grammar. We do so as follows: for a non-probabilistic grammar, all sizes are equal to 1 and for a probabilistic grammar, sizes are equal to the normalized negative logarithmic probabilities:

$$size(R) = Round\left(\frac{-\log(p_R)}{\min_{R' \in \mathcal{R}(N_R)} (-\log(p_{R'}))}\right)$$

where N_R is the nonterminal which produces the rule R and *Round* is the function rounding to the nearest integer. We normalize probabilities to reduce the size of the operators, which reduces the maximum size of terms that STE has to explore for a given benchmark. In any case, STE was designed for non-probabilistic grammars, and we do not evaluate it for probabilistic grammars.

3.3 Probabilistic Top-Down Term Enumeration

In the search phase of the CEGIS loop, we want to find an expression T that satisfies the requirement in Equation 3.1. One approach to implementing $\text{Search}(\bar{a}, A, \Pi, \phi, x)$ is to enumerate all candidate expressions, in some order, until a suitable answer is found. Algorithm 3.4 presents this approach, parametric to the `Enumerate` procedure. Recall that, because A is a finite set, for a given choice of A and T , determining the satisfaction of Equation 3.1 reduces to evaluating the conjunction and does not involve expensive calls to an SMT solver. The enumerative SyGuS solver [ABJ⁺13] uses a highly optimized form of expression enumeration in the search phase, and is currently among the most competitive SyGuS solvers.

Algorithm 3.4: $\text{Search}(\bar{a}, A, \Pi, \phi, x)$. Implements the search phase of the CEGIS loop by expression enumeration.

- (1) Let G be the chosen term grammar, and \mathcal{S} be the starting nonterminal of the same type as the output variable x , i.e., $T_{\mathcal{S}} = T_x$.
 - (2) For each e emitted by $\text{Enumerate}(G, N)$:
 - (a) If $\bigwedge_{\bar{a} \in A} [\Pi \rightarrow \phi[x \mapsto e] = \text{true}]$, return e .
 - (b) Otherwise, discard e and continue enumeration.
-

In the rest of this section, we describe an implementation of `Enumerate` that uses a probabilistic term grammar to enumerate expressions in order of decreasing probability. This way, it can discover the solution with the maximum probability, as well as accelerate the search process.

3.3.1 Derivation Trees

We first extend the idea of a grammar derivation into the more general notion of a *partial* derivation:

$$\tilde{e}_N ::= ?_N \mid \mathbf{t}(\tilde{e}_{N_1}, \tilde{e}_{N_2}, \dots, \tilde{e}_{N_k}),$$

where $R = \mathbf{t}(N_1, N_2, \dots, N_k)$ is a derivation rule of N in G . In particular, note that partial derivations can contain empty sub-expressions, denoted by $?_N$. We will sometimes omit the type of empty subexpressions. Examples of partial derivations include “ $x + ?$ ” and “if ($x \leq ?$) { x } else { $? \}$ ”. We write $\tilde{\mathcal{E}}_N$ for the set of all partial derivations of a nonterminal N .

Given a pair of partial derivations \tilde{e}_1 and \tilde{e}_2 , we say that \tilde{e}_1 and \tilde{e}_2 are related by the expansion relation, and write $\tilde{e}_1 \rightarrow \tilde{e}_2$, if \tilde{e}_2 can be obtained by replacing *the left-most instance* of $?$ in \tilde{e}_1 by an appropriate production rule. For example, if $\tilde{e}_1 = x + ?$, $\tilde{e}_2 = x + 1$, and $\tilde{e}_3 = x + (? + ?)$, then $\tilde{e}_1 \rightarrow \tilde{e}_2$ and $\tilde{e}_1 \rightarrow \tilde{e}_3$.

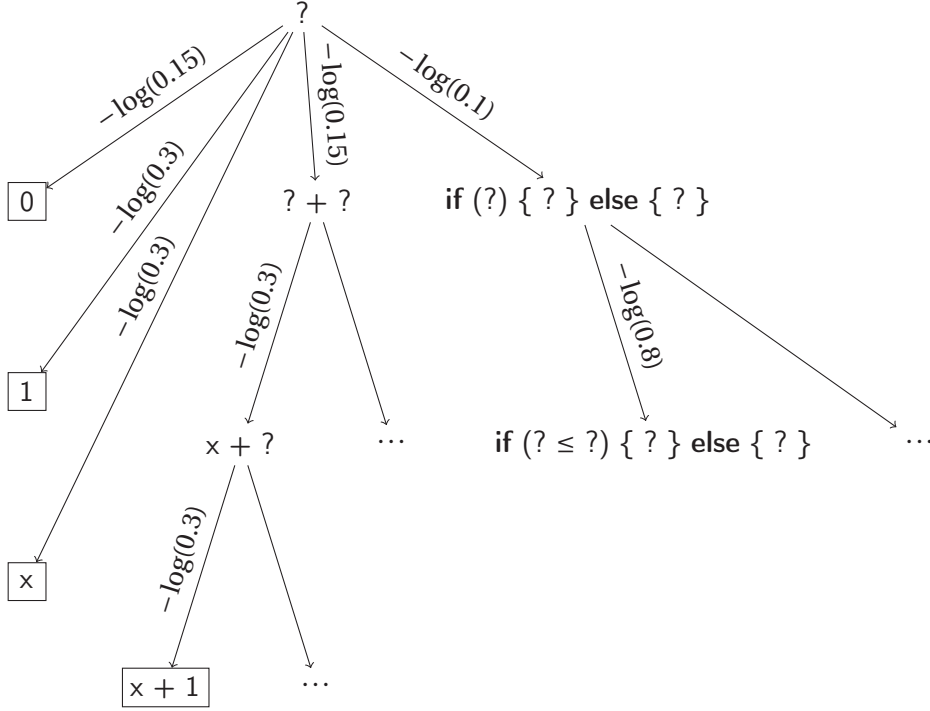


Figure 3.1 – A tree of partial derivations for a PCFG

The expansion relation naturally induces a tree \mathcal{G} on the partial derivations of a grammar G , as displayed in Figure 3.1. The initial node is the starting nonterminal, $?_{\mathcal{G}}$. There is an edge from the partial derivation \tilde{e}_1 to the partial derivation \tilde{e}_2 if they are related by the expansion relation, $\tilde{e}_1 \rightarrow \tilde{e}_2$. Because we can only replace the left-most instance of $?$, the resulting graph is a tree. For clarity, complete derivations are squared in the figure.

We can also speak of the probability of partial derivations: $p_{\tilde{e}}$ of a partial derivation \tilde{e} is the product of the probabilities of all production rules used to derive \tilde{e} . Those rules are a multiset $r(\tilde{e})$, defined as follows: $r(?) = \emptyset$ and $r(\mathbf{t}(\tilde{e}_{N_1}, \tilde{e}_{N_2}, \dots, \tilde{e}_{N_k})) = R \cup \bigcup_i r_{N_i}$, where $R = \mathbf{t}(\tilde{e}_{N_1}, \tilde{e}_{N_2}, \dots, \tilde{e}_{N_k})$. Thus, $p_{\tilde{e}} = \prod_{R \in r(\tilde{e})} p_R$.

It is mathematically more convenient to speak of negative logarithmic probabilities: the *cost* of a rule R is defined as $-\log(p_R)$, and the cost of a partial derivation \tilde{e} ,

$$\text{cost}(\tilde{e}) = -\log(p_{\tilde{e}}) = - \sum_{R \in r(\tilde{e})} \log(p_R), \quad (3.2)$$

In Figure 3.1, the edge $\tilde{e}_1 \rightarrow \tilde{e}_2$, produced by an instantiation of the rule R , is annotated with the cost of the rule, $-\log(p_R)$. The sum of the weights along the path to \tilde{e} is equal to $\text{cost}(\tilde{e})$. A main insight in this section is that each $\text{Enumerate}(G, N)$ can be thought of as an instantiation of a search algorithm on this tree.

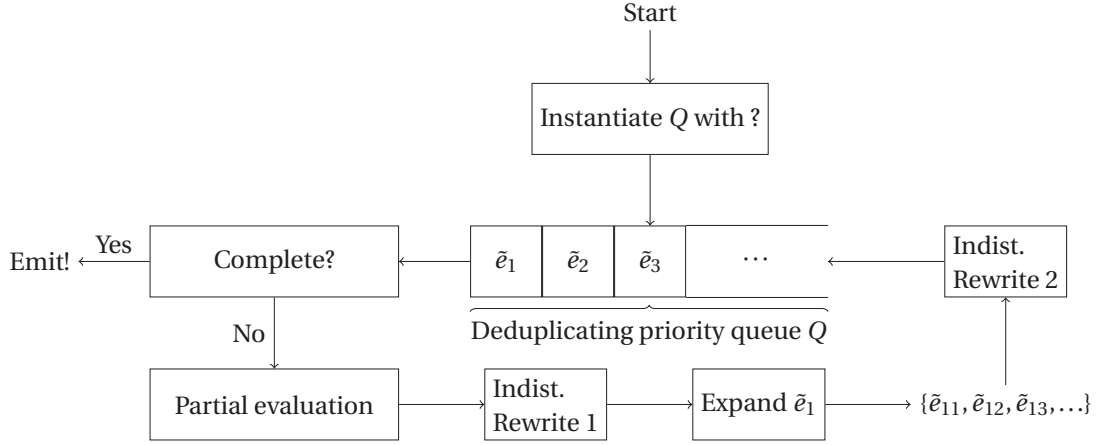


Figure 3.2 – The operation of Enumerate(G, N) in Algorithm 3.5

3.3.2 The Enumerate Algorithm

Algorithm 3.5: Enumerate(G, N). Emits a sequence of complete derivations of N .

- (1) Let $\pi : \tilde{\mathcal{E}}_N \rightarrow \mathbb{R}_{\geq 0}$ be a priority function that maps derivations to non-negative real numbers.
- (2) Let Q be a priority queue of partial derivations $\tilde{e} \in \tilde{\mathcal{E}}_N$ arranged in ascending order according to π . Initialize $Q := \{?_N\}$.
- (3) While Q is not empty:
 - (a) Let \tilde{e} be the element at the front of Q . Dequeue \tilde{e} .
 - (b) If \tilde{e} is a complete derivation, emit \tilde{e} .
 - (c) Otherwise, for every neighbor \tilde{e}' such that $\tilde{e} \rightarrow \tilde{e}'$ in G , insert \tilde{e}' into Q .

In this section, we present and analyze an instantiation of Enumerate based on probabilistic term grammars, depicted in Algorithm 3.5. It is helpful to visualize the operation of the algorithm as shown in Figure 3.2. The algorithm maintains a priority queue Q of still-unexpanded partial derivations. At each step, the algorithm dequeues the element \tilde{e} at the front of this priority queue and, if it is still incomplete, expands the leftmost $?$ with an instance of every applicable production rule R . This results in a set of partial derivations $\tilde{e}_1, \tilde{e}_2, \dots, \tilde{e}_k$. All of these new partial derivations are inserted back into Q . If \tilde{e} is already complete, it is emitted to be further processed by Search($\tilde{a}, A, \Pi, \phi, x$). If $\pi(\tilde{e}) = \text{cost}(\tilde{e}) = -\log(P(\tilde{e}))$, i.e., the priority function is equal to the cost of the partial derivation, then Enumerate emulates Dijkstra's algorithm.

We begin analysing Algorithm 3.5 with the following invariant:

Theorem 3.1. Let \mathcal{E}_c be the set of (complete) derivations already emitted by Enumerate(G, N).

Then, $\sum_{\tilde{e} \in Q \cup \mathcal{E}_c} P(\tilde{e}) = 1$.

Proof. First, observe that, for all partial derivations \tilde{e} with at least one occurrence of $?$,

$$\sum_{\tilde{e}' \text{ s.t. } \tilde{e} \rightarrow \tilde{e}'} P(\tilde{e}') = P(\tilde{e}). \quad (3.3)$$

This follows directly from Definition 2.4. Informally, if each partial derivation \tilde{e} is viewed as an event resulting from the PCFG, then the partial derivations \tilde{e}' obtained by a single application of the expansion relation encode a mutually exclusive and exhaustive collection of sub-events of \tilde{e} .

We then have: (1) the base case: $P(?) = 1$, (2) inductive case #1: whenever a complete derivation e is emitted from Q , the set $Q \cup \mathcal{E}_c$ is left unchanged, (3) and inductive case #2: whenever a partial derivation \tilde{e} is further expanded into partial derivations $\tilde{e}_1, \tilde{e}_2, \dots, \tilde{e}_k$, the candidate invariant continues to hold because of Equation 3.3. \square

Intuitively, Theorem 3.1 captures the fact that each possible complete derivation either has already been discovered (belongs to \mathcal{E}_c), or is derivable from a partial derivation in Q ; therefore, the probabilities of these derivations sum to 1.

Theorem 3.2. *If the priority function, $\pi(\tilde{e}) = \text{cost}(\tilde{e}) = -\log(P(\tilde{e}))$, then $\text{Enumerate}(G, N)$ returns some sequence σ containing all complete derivations of N in G . Furthermore, the probability $P(e)$ of derivations in σ is monotonically decreasing.*

Proof. We begin by proving the second part of the claim. Consider a pair of partial derivations, \tilde{e} and \tilde{e}' such that $\tilde{e} \rightarrow \tilde{e}'$. Then, $P(\tilde{e}) \geq P(\tilde{e}')$. In other words, every derivation that is dequeued will only enqueue derivations of lesser probabilities into Q . It therefore follows that the sequence of partial derivations, τ , dequeued by $\text{Enumerate}(G, N)$ in line (a) has monotonically decreasing probability. Observe that the sequence σ of complete derivations emitted by $\text{Enumerate}(G, N)$ is a sub-sequence of τ . Therefore, the probability of derivations in σ monotonically decreases.

Now, arbitrarily choose a complete derivation $e \in \mathcal{E}_N$. We show that $\text{Enumerate}(G, N)$ will eventually emit e . First, we show that if e gets enqueued in Q , it will eventually be dequeued, by showing there are finite many derivations with probability greater than e . Let p_{hi} be the highest probability of any production rule in G . Recall our requirement that for each rule R , $p_R < 1$, therefore, $p_{\text{hi}} < 1$. Every derivation e' with probability $P(e') \geq P(e)$ must contain $n \leq \log(P(e)) / \log(p_{\text{hi}})$ production expansions. Otherwise,

$$P(e') = \prod_{R \in e'} p_R \leq p_{\text{hi}}^n < p_{\text{hi}}^{\log(P(e)) / \log(p_{\text{hi}})} = P(e)$$

Then, because the number of production rules is finite, there is a finite number for those derivations.

Chapter 3. Term Exploration

Given this lemma, we can prove that e is eventually emitted by a simple structural inductive argument on the expansion relation. \square

The first property stated in Theorem 3.2 demonstrates the completeness of Algorithm 3.5 for the specific choice of π , in the sense that it will eventually derive every complete derivation (language term) that is derivable by the input grammar G . Note that, if we additionally assume that the Verify part of Algorithm 3.1 always returns the correct result, and that the grammar G derives at least one term that satisfies the specification ϕ , then the Algorithm 3.1 is complete.

The second property is an optimality property of Algorithm 3.5 in the following sense: assume that a term satisfies the specification ϕ with a probability identical to the probability of its derivation from G ; then, from the fact that the most probable complete derivations get enumerated first, it is easy to show that the search will enumerate on average the least possible number of complete derivations before discovering the satisfactory term.

Despite this optimality property regarding complete derivations, when using the priority function π given above, enumeration does not scale well. The intuition can be found in the correctness proof above: before emitting e , the enumerator must explore all *partial* derivations with probability bigger than $P(e)$. The number of these derivations rapidly grows with decreasing $P(e)$. In addition, most of the processed partial derivations are “very incomplete”, i.e., they contain many instances of $?$ and therefore are many edges away from turning into complete derivations. Note that, with this instantiation of π , Enumerate emulates Dijkstra’s algorithm on the graph of partial derivations. The lack of performance of this instantiation suggests the use of other graph search algorithms, which we will now discuss.

3.3.3 Nonterminal Horizons and A* Search

When considering partial derivations \tilde{e} , we can speak of two quantities: (1) the cost *already paid*, which we have defined as $\text{cost}(\tilde{e}) = -\log(P(\tilde{e}))$, and (2) the minimum cost *yet to be paid*, before \tilde{e} turns into a complete derivation. We formalize this latter quantity as the *horizon*, defined (non-recursively) as

$$h(\tilde{e}) = \left(\min_{e_f \text{ s.t. } \tilde{e} \rightarrow^* e_f} \text{cost}(e_f) \right) - \text{cost}(\tilde{e}), \quad (3.4)$$

where e_f ranges over all complete derivations reachable from \tilde{e} . Recall the intuition for the practical failure of Dijkstra’s algorithm: Before emitting e , we must process every partial derivation with higher probability, and most such partial derivations \tilde{e}' are themselves many steps away from complete derivations. As the horizon encodes the distance from the partial derivation to its nearest completion, it is natural to include it in the priority function $\pi(\tilde{e})$. If we define $\pi(\tilde{e}) = \text{cost}(\tilde{e}) + h(\tilde{e})$, then we obtain A* search.

The important property of $\pi(\tilde{e})$ is that it is *admissible*, i.e., that $\pi(\tilde{e})$ is always less or equal to the cost of every complete derivation descendant from \tilde{e} . More formally, for all complete

derivations e_c reachable from \tilde{e} ,

$$\pi(\tilde{e}) = \min_{e_c \text{ s.t. } \tilde{e} \rightarrow^* e_c} \text{cost}(e_c). \quad (3.5)$$

We then have the following well-known result of the A* algorithm:

Theorem 3.3. *If $\pi(\tilde{e}) = \text{cost}(\tilde{e}) + h(\tilde{e})$, then:*

- (1) *The sequence of complete derivations emitted by Enumerate(G, N), $\sigma = e_1, e_2, e_3, \dots$ contains all complete derivations in \mathcal{E}_N and has monotonically decreasing probability. (This is called the optimality property.) Additionally, the sequence of all derivations dequeued from Q contains all derivations in \mathcal{E}_N and has monotonically decreasing probability.*
- (2) *There exists no complete optimal algorithm $A'(G, N)$ that, given a complete derivation e , will visit less nodes than A* before producing e . Those are all strictly less expensive nodes, \tilde{e} , such that $\pi(\tilde{e}) < \pi(e)$. (This is the property of optimal efficiency.)*

Proof. First, we show that Enumerate(G, N) produces all complete derivations $e \in \mathcal{E}_N$. Similarly as in Theorem 3.2, we demonstrate the finiteness of the number of nodes that can be processed in line (a) before processing e . Observe that for all nodes, \tilde{e} , $\pi(\tilde{e}) \geq \text{cost}(\tilde{e})$, and for complete derivations e , $\pi(e) = \text{cost}(e)$. Hence as before, every derivation with probability $\geq P(e)$ must contain at most $\log(P(e))/\log(p_{\text{hi}})$ production expansions, and the reasoning of Theorem 3.2 holds. Observe that the proof does not have to be constrained to complete derivations, with which we prove the second part of the first point.

We will now show that the derivations dequeued have monotonically decreasing probabilities. For this, first observe that for each complete derivation e , and for each of its (necessarily incomplete) ancestors \tilde{e}' , $\pi(\tilde{e}') \leq \pi(e)$. For the sake of contradiction, let there be some pair e_1, e_2 of complete derivations, such that $P(e_1) < P(e_2)$, but Enumerate(G, N) dequeues e_1 before e_2 . Both e_1 and e_2 are therefore reachable from the initial node $?$. Consider the state of Q when e_1 is dequeued from it. It has to be the case that some ancestor \tilde{e}'_2 is present in Q . However, it follows that $\pi(\tilde{e}'_2) \leq \pi(e_2) < \pi(e_1)$, therefore the priority queue must have made a mistake in its ordering. It follows that the sequence of derivations dequeued by Enumerate(G, N) has monotonically increasing costs, or equivalently, monotonically decreasing probabilities. Note that this means that A* visits all strictly less expensive nodes, \tilde{e} , such that $\pi(\tilde{e}) < \pi(e)$ before visiting e . Since the sequence of emitted complete derivations is a subsequence of dequeued derivations, this proves optimality of A*.

We will now prove the last part of the theorem. Assume otherwise: Let there be an algorithm $A'(G, N)$ and some node \tilde{e} such that $\pi(\tilde{e}) < \pi(e)$, A' does not expand \tilde{e} before producing e . We know \tilde{e} has some descendant e' such that $\pi(\tilde{e}) = \text{cost}(e') < \pi(e)$. Because the search space \mathcal{G} is a tree, if A' does not expand \tilde{e} , it follows that it did not enumerate e' before e , violating the assumption that A' is a complete optimal enumerator. \square

The first part of Theorem 3.3 states the completeness and optimality of the A* algorithm for synthesis, similar to how Theorem 3.2 does for the Dijkstra version. However, optimality as stated here is stronger, in the sense that less partial derivations are expanded. This is because the priority function π chosen here is equal to the Dijkstra version for complete derivations, but larger for partial derivations. This captures our intuition for the superiority of this priority function over the simpler priority function used by Dijkstra's algorithm. Furthermore, the second part of Theorem 3.3 states the superiority of A* over all other algorithms that use the same priority function.

3.3.4 Computing the Horizon, $h(\bar{e})$

It can be shown that the minimum cost to be paid to complete \bar{e} is the sum of the minimum costs needed to expand each of its unexpanded nodes:

$$h(\bar{e}) = \sum_{?_N \in \bar{e}} h(?_N). \quad (3.6)$$

Before starting the enumerator, we therefore compute the nonterminal horizons, $h(?_N)$, for each nonterminal symbol N . Observe that, by definition, this simply encodes the probability of the most likely derivation, $e \in \mathcal{E}_N$:

$$h(?_N) = \min_{e \in \mathcal{E}_N} \text{cost}(e) = -\max_{e \in \mathcal{E}_N} \log(P(e)).$$

The most likely derivation can be computed efficiently with a specialization [CdIH00, NS08] of an algorithm by Knuth [Knu77]. When applied on a grammar $G = (\mathcal{N}, \mathcal{R}, \mathcal{S})$, this algorithm runs efficiently in $O(|\mathcal{R}| \cdot \log |\mathcal{N}| + L)$ time, where L is the total length of all rules in \mathcal{R} [Knu77].

3.3.5 Optimizations

Eagerly discarding partial productions. Given a single input variable a , consider the synthesis predicate ϕ given by

```

if (a == 5) { x == 6 }
else if (a == 7) { x == 9 }
else { x == a }

```

During CEGIS, say the set of concrete input points, $A = \{2, 5, 7\}$. For this problem, $\text{Enumerate}(G, N)$ will consider partial productions such as

$$\bar{e} = \text{if } (x < 6) \{x\} \text{ else } \{?\}.$$

Observe that the conjunction $\bigwedge_{a \in A} \phi[x \mapsto e]$ used in the CEGIS loop can be evaluated only for complete productions e . However the partial production \bar{e} is already incorrect: for the

input point $a = 5$, regardless of the completion of $?$, \tilde{e} will evaluate to 5, and this fails the requirement that $x == 6$.

This pattern of partial productions that already fail requirements is particularly common with conditionals and match statements. Our first optimization is therefore the partial evaluation box shown in Figure 3.2. For each input point $\tilde{a} \in A$, we partially evaluate the CEGIS predicate $\Pi \Rightarrow \phi[x \mapsto \tilde{e}]$. If the result is **false**, then we discard the partial production \tilde{e} without processing, as we know that all its descendants will fail the synthesis predicate. Otherwise, if it evaluates to **true** or **unknown**, then we process \tilde{e} as usual, and insert its neighbors back into the priority queue.

Extending the priority function with scores. We just observed that if a partial production fails the synthesis predicate by partial evaluation, then it can be discarded without affecting correctness. The dual heuristic optimization is to *promote* partial productions that definitely evaluate to **true** on some input points. We do this by modifying the priority function

$$\pi(\tilde{e}) = \min(0, \text{cost}(\tilde{e}) + h(\tilde{e}) - c \times \text{score}(\tilde{e})), \quad (3.7)$$

where $\text{score}(\tilde{e})$ is the ratio of input points on which the partial production evaluates to **true** over all available input points, and c is a positive coefficient. Small positive values of c results in an enumerator that strictly follows probabilistic enumeration, whereas large positive values of c results in the enumerator favoring partial productions that already work on many points. Although the use of this heuristic renders Theorem 3.3 inapplicable, we have observed improvements in performance as a result of this optimization for benchmarks involving conditional expressions.

Indistinguishability. Consider the partial production $\tilde{e} = \text{if } (x < 5) \{x - x\} \text{ else } \{?\}$, and focus on the sub-expression $x - x$. This expression is identically equal to 0, and it is therefore possible to simplify \tilde{e} to $\text{if } (x < 5) \{0\} \text{ else } \{?\}$. If two expressions e and e' are equivalent, and $P(e) > P(e')$, then a desirable optimization is to never enumerate e' . Observe that, if properly implemented, this optimization produces exponential savings at each step of the search: If e and e' are equivalent, then, for example, it follows that both $e + 3$ and $e' + 3$ are equivalent, and only one of them needs to be enumerated to achieve completeness.

Indistinguishability [URD⁺13] is a technique for mechanizing this reasoning. Given a sequence of concrete input points $A = \{\tilde{a}_1, \tilde{a}_2, \dots, \tilde{a}_n\}$, a complete production e can be evaluated to produce a set of output values $S_e = \{x_1, x_2, \dots, x_n\}$. During enumeration, if an expression e' is encountered such that for some previous expression e , $S_e = S_{e'}$, then e' is discarded as a potential expansion. The original expression e can thus be regarded as the representative of the equivalence class of all expressions whose signature is equal to S_e .

In this original formulation of indistinguishability-based pruning, the enumerator worked

bottom-up, rather than in the top-down style we consider in this dissertation. As a result, the enumerator needs to process only complete productions, rather than the partial productions that populate Q in our setting.

We extend this optimization to work with partial expressions, and to prune expressions in a top-down enumerator. The idea is to replace the priority queue Q with a “deduplicating priority queue”, with an additional feature that Q removes duplicate elements from being dequeued. Such a queue can be implemented as a combination of a traditional priority queue and a mutable set data structure commonly available in standard language libraries. Next, we maintain a dictionary that maps all previously seen signatures to the representatives of the respective equivalence classes. Every time we dequeue a partial production \tilde{e} from Q , we evaluate every complete sub-production e_{sub} on all input points $\tilde{a} \in A$ to obtain its signature S_{sub} , and replace e_{sub} with the canonical representative. This forms the box labelled “Indist. rewrite 1” in Figure 3.2.

After each partial production is expanded, we perform a second, fast indistinguishability rewrite. This is done by maintaining a map from previously seen expressions to their canonical representatives. Consulting this map for every complete sub-production of the children, $\tilde{e}_1, \tilde{e}_2, \dots$, of the original production \tilde{e} is much faster than evaluating them on each concrete input point in A . There is some freedom in choosing the placement of various optimizations in Figure 3.2: Our motivation was that the priority queue can become very large, and many elements enqueued into the queue will never be dequeued, and that it is therefore wise to postpone as much processing as possible to when partial productions are extracted from Q .

As would be expected, term exploration with indistinguishability scales poorly when a large number of examples is used, due to the large amount of evaluation involved. Therefore, we limit the number of examples automatically generated by the system when deploying indistinguishability.

3.4 Evaluation

We evaluate synthesis on a set of benchmarks that mostly manipulate functional data structures. All benchmarks and their solutions can be found in Appendix A and B.

3.4.1 Aims

The aims of our evaluation are

- (1) to evaluate and compare the efficiency of the two different algorithms presented in the previous sections,
- (2) in the case of Probabilistic Term Enumeration (PTE), to investigate which flavor of term grammars leads faster to a solution. The first one we evaluate is the built-in grammar of

Leon described in Section 2.5. Remember that this is not a probabilistic grammar, i.e., all probabilities assigned to the production rules of a nonterminal symbol are equal. The other two grammars we evaluate are probabilistic grammars extracted from a corpus of programs, as described in Section 2.7.3.

- (3) to evaluate the effectiveness of the optimizations suggested in Section 3.3.5.

3.4.2 Description of Tables

Synthesis from specification. In Tables 3.1 and 3.2, we give for each benchmark

- the total size of the program, excluding data structures defined in the default Leon library,
- the size of the synthesized solution and
- for each of four configurations that are explained below, the time it took to synthesize the solution, and whether the solution was verifiable with the Leon verifier. An X means that synthesis was unsuccessful.

The difference between the two tables are the following: in Table 3.1, the Leon deductive rules are instantiated automatically. Hence we evaluate the enumeration algorithms in tandem with the Leon synthesis framework. In Table 3.2, deductive rules are instantiated manually, and so the enumeration algorithms are evaluated in isolation from the deductive synthesis framework. In both cases, the deductive rules generate the necessary if-conditions, pattern matches and recursive calls. Therefore, the term grammars used in the enumeration algorithms do not produce such expressions. This is true both for the built-in and extracted grammars.

We test the following four configurations:

- Symbolic Term Exploration (STE) with the built-in grammar (STE, 0)
- PTE with the built-in grammar (PTE, 0)
- PTE with plain probabilistic extracted grammar (PTE, 1) and
- PTE with annotated probabilistic extracted grammar (PTE, 2)

For all but the annotated probabilistic grammars, we inject aspects into the grammar nonterminals.

Synthesis by symbolic examples. Next, we test our technique on a few benchmarks where the specification is given as symbolic examples. We ran the same configurations as above except STE, as it is not optimized to handle these benchmarks. The results are listed in Table 3.3.

Evaluating optimizations. The next few tables are dedicated to individually evaluate each optimization of Section 3.3.5, as well as aspect grammars (Section 2.2).

In Table 3.4, we compare running times of PTE with or without indistinguishability heuristic. Benchmarks are run in automatic mode with built-in grammars. In Table 3.5, we demonstrate the effectiveness of aspect grammars in reducing program spaces. Benchmarks are run with STE in automatic mode. To make the effect of aspects more clear, we also indicate how many programs were considered by STE in each configuration.

To evaluate the scores and early discarding of erroneous candidates, we have to delegate the synthesis of **if**-conditions to the enumeration rules rather than the splitting rules of the deductive framework. We took all benchmarks of Table 3.1 that need the synthesis of **if**-conditions and ran in manual mode, instantiating PTE where an **if**-clause is required. We used a grammar extracted from corpus but, in contrast with the previous experiments, we did not delete **if**-conditions. The results are presented in Table 3.6. The timeout for this set of benchmarks is 300 seconds.

3.4.3 Assessment of Results

Both algorithms are able to efficiently synthesize a variety of functional functions. In particular, benchmarks like `RunLength` are particularly hard to solve and outside the scope of most synthesis tools, at least when a large number of possible program components are considered. Another tool that can solve this benchmark is the SYNQUID system [PKS16]. However, the version of the benchmark for their system that is available online [Syn] explicitly lists the zero and one constants as well as the successor, predecessor and equality functions as the only primitive building blocks for arithmetic expressions. In contrast, our system explores trees that, in addition to constants, contain general binary arithmetic operations including addition and subtraction operators. As a result, our search space is notably larger. In our attempts, adding components corresponding to our search space made the SYNQUID web example timeout after the 120 second limit. A more systematic comparison of the two tools remains to be done in the future.

To compare the two algorithms, let us look at Table 3.2. PTE clearly outperforms STE, with `BatchedQueue.dequeue` being the only meaningful exception. The grammars extracted from corpus show no significant improvement, also except `BatchedQueue.dequeue`. Additionally, the annotated grammars fail to synthesize the `RunLength` benchmark. This is because some combination of parent-child tree required to synthesize it was not found in the corpus. This hints to the fact that we need a more complicated statistical model of the corpus incorporated in our term grammars, or a bigger corpus to obtain more reliable probabilities.

In Table 3.1, where the whole deductive synthesis framework comes into play, we can see that the advantage of PTE is less pronounced, and even performs less efficiently for some benchmarks. The reason for that is that PTE takes more time to try to synthesize hopeless





Operation	Sizes		STE,0		PTE,0		PTE,1		PTE,2	
	Pr	Sol		⊢		⊢		⊢		⊢
List.insert	81	3	1.5	✓	0.4	✓	0.4	✓	0.4	✓
List.delete	83	19	15.1	✓	19.1	✓	24.9	✓	12.9	✓
List.union	81	12	5.6	✓	5.2	✓	1.1	✓	1.3	✓
List.diff	113	12	9.4	✓	9.7	✓	1.8	✓	2.3	✓
List.split	104	20	3.5	✓	5.5	✓	1.4	✓	1.6	✓
List.listOfSize	58	11	3.0	✓	2.5	✓	5.1	✓	3.2	✓
SortedList.insert	114	30	21.0	?	30.2	?	39.1	?	21.5	?
SortedList.insertAlways	128	32	25.3	✓	38.8	✓	48.4	✓	27.1	✓
SortedList.delete	114	19	20.4	?	26.2	?	31.3	?	39.6	?
SortedList.union	162	12	5.9	✓	6.1	✓	3.1	✓	6.8	✓
SortedList.diff	160	12	8.1	✓	9.3	✓	8.7	✓	11.3	✓
SortedList.insertionSort	149	11	3.0	✓	4.8	✓	1.8	✓	5.5	✓
StrictSortedList.insert	114	30	25.4	✓	28.5	✓	32.9	✓	19.6	✓
StrictSortedList.delete	114	19	24.2	✓	26.2	✓	37.5	✓	30.9	✓
StrictSortedList.union	162	12	7.5	✓	6.2	✓	3.1	✓	7.1	✓
UnaryNumerals.add	66	10	5.7	✓	1.0	✓	1.1	✓	1.3	✓
UnaryNumerals.distinct	91	4	2.4	✓	0.7	✓	0.5	✓	0.9	✓
UnaryNumerals.mult	66	11	4.2	✓	5.1	✓	0.9	✓	5.2	✓
BatchedQueue.enqueue	112	26	16.7	?	16.1	?	12.2	?	12.6	?
BatchedQueue.dequeue	88	35	19.8	?	14.5	?	8.7	✓	64.9	✓
AddressBook.makeAddressBook	63	33	8.6	✓	6.6	✓	4.8	✓	4.1	✓
AddressBook.merge	126	17	20.9	?	19.1	?	16.8	?	X	X
RunLength.encode	138	39	28.1	✓	21.8	✓	23.0	✓	X	X
Diffs.diffs	83	24	9.5	✓	11.7	✓	9.5	✓	11.3	✓

Table 3.1 – Benchmarks for synthesis in automatic mode

problems which arise during instantiation of the deductive rules, whereas STE gives up faster in those cases. It is a topic of further work to integrate the probabilistic algorithm better into the framework to avoid this effect.

In Table 3.3, we demonstrate that Leon can also efficiently solve benchmarks where the specification is given by symbolic or concrete examples. For example, look at the `Diffs.diff` benchmark, which computes the running differences of an integer list from three examples:

```
def diffs(l: List[BigInt]): List[BigInt] = {
  ???[List[BigInt]]
} ensuring { (res: List[BigInt]) ⇒ (l, res) passes {
  case Nil() ⇒ Nil()
  case Cons(BigInt(55), Nil()) ⇒ List(55)
  case Cons(BigInt(100), Cons(BigInt(-100), Nil())) ⇒ List(200, -100)
  case Cons(BigInt(1), Cons(BigInt(2), Cons(BigInt(22), Nil()))) ⇒ List(-1, -20, 22) } }
```


Chapter 3. Term Exploration





Operation	Sizes		STE,0		PTE,0		PTE,1		PTE,2	
	Pr	Sol		⊢		⊢		⊢		⊢
List.insert	81	3	1.4	✓	0.3	✓	0.2	✓	0.3	✓
List.delete	83	19	5.0	✓	0.8	✓	0.8	✓	1.1	✓
List.union	81	12	2.6	✓	0.4	✓	0.4	✓	0.5	✓
List.diff	113	12	2.9	✓	1.1	✓	0.5	✓	0.6	✓
List.split	104	20	2.2	✓	1.6	✓	1.0	✓	0.8	✓
List.listOfSize	58	11	1.5	✓	0.5	✓	0.5	✓	0.4	✓
SortedList.insert	114	30	8.6	?	7.0	?	11.3	?	6.4	?
SortedList.insertAlways	128	32	12.2	✓	11.6	✓	11.3	✓	11.5	✓
SortedList.delete	114	19	11.5	?	6.3	?	6.2	?	6.3	?
SortedList.union	162	12	3.2	✓	0.8	✓	1.0	✓	1.0	✓
SortedList.diff	160	12	3.1	✓	1.0	✓	1.2	✓	1.2	✓
SortedList.insertionSort	149	11	1.8	✓	0.5	✓	0.8	✓	0.4	✓
StrictSortedList.insert	114	30	10.0	✓	6.7	✓	6.3	✓	6.4	✓
StrictSortedList.delete	114	19	10.1	✓	6.5	✓	6.2	✓	6.4	✓
StrictSortedList.union	162	12	3.7	✓	0.9	✓	1.2	✓	1.2	✓
UnaryNumerals.add	66	10	3.6	✓	0.4	✓	0.3	✓	0.5	✓
UnaryNumerals.distinct	91	4	2.3	✓	0.6	✓	0.8	✓	1.2	✓
UnaryNumerals.mult	66	11	3.2	✓	0.5	✓	0.5	✓	0.6	✓
BatchedQueue.enqueue	112	26	7.6	?	6.4	?	11.5	?	11.6	?
BatchedQueue.dequeue	88	35	7.3	?	30.8	?	4.0	✓	7.6	✓
AddressBook.makeAddressBook	63	33	4.8	✓	2.0	✓	2.1	✓	2.4	✓
AddressBook.merge	126	17	21.1	?	29.2	?	28.5	?	34.4	?
RunLength.encode	138	39	11.7	✓	4.6	✓	5.9	✓	10.7	X
Diffs.diffs	83	24	5.2	✓	3.9	✓	1.9	✓	4.6	✓

Table 3.2 – Benchmarks for synthesis in manual mode

```
// Solution:
def diffs(l : List[BigInt]): List[BigInt] = {
  l match {
    case Nil() =>
      List[BigInt]()
    case Cons(h, t) =>
      t match {
        case Nil() =>
          List(h)
        case Cons(h1, t1) =>
          Cons[BigInt](h - h1, diffs(t1)) } } }
```

In the final three tables, we see that all suggested optimizations have positive impact on synthesis. Indistinguishability reduces running times by a factor of two or more. Aspect grammars, while having a large effect on the number of considered programs, do not display




Operation	Sizes		PTE,0		PTE,1		PTE,2	
	Pr	Sol		⊢		⊢		⊢
UnaryNumerals.add	63	10	2.9	✓	0.5	✓	0.9	✓
List.append	112	12	0.7	✓	0.5	✓	0.9	✓
Calc.eval	83	49	4.5	✓	4.7	✓	14.2	✓
Tree.countLeaves	69	13	0.7	✓	0.6	✓	19.4	✓
Dictionary.replace	199	26	8.4	✓	2.7	✓	3.1	✓
Dictionary.find	138	18	1.1	✓	0.9	✓	1.5	✓
List.diffs	93	22	12.9	✓	15.9	✓	10.6	✓
Expr.fv	93	36	2.1	✓	7.0	✓	10.9	✓
UnaryNumerals.isEven	46	9	2.0	✓	3.4	✓	3.5	✓
SortedList.insert	142	30	25.6	✓	40.8	✓	14.1	✓
UnaryNumerals.mult	86	11	8.1	✓	0.7	✓	8.2	✓
Tree.postorder	95	15	0.9	✓	1.0	✓	1.7	✓
List.reserve	99	13	42.3	✓	0.9	✓	90.7	✓
RunLength.encode	207	39	41.2	✓	12.2	✓	28.6	✓
List.take	110	19	5.5	✓	3.4	✓	3.9	✓
List.unzip	103	24	0.8	✓	0.5	✓	0.9	✓

Table 3.3 – Benchmarks for synthesis by example in automatic mode

the same improvement in the running time, due to the cost associated with dynamically generating and handling more complicated grammars. Unintuitively, aspect grammars do not combine well with PTE, so comparative results for aspects with PTE are not shown. Some of the results of Table 3.6 are significantly worse than Table 3.2 because the system has to explore a much larger space of programs, specifically when it comes to conditions of if-expressions, as opposed to examining a small set produced by the deductive rules. However, Table 3.6 clearly demonstrates the strong impact of scores when synthesizing programs with if-conditions. This is not to say that scores would benefit every benchmark, but they work well in our setting.

3.5 Conclusion

In this chapter, we presented two term exploration algorithms. The first, named Symbolic Term Exploration, is an evolution of a previous algorithm of Leon. It unfolds a given grammar in order of increasing term size and exhaustively explores terms of a specific size before moving to the next. The second, named Probabilistic Term Enumeration, creates partial derivation trees corresponding to incomplete expressions, then completes the holes in these expressions in best-first order. Probabilistic Term Enumeration also incorporates a number of domain-specific optimizations. We then extensively evaluate these algorithms, as well as the described optimizations and different variants of term grammars.

Chapter 3. Term Exploration

Operation	Sizes		Indist. off	Indist. on
	Pr	Sol		
List.insert	81	3	0.4	0.4
List.delete	83	19	44.0	19.1
List.union	81	12	18.7	5.2
List.diff	113	12	36.0	9.7
List.split	104	20	26.8	5.5
List.listOfSize	58	11	2.3	2.5
SortedList.insert	114	30	64.8	30.2
SortedList.insertAlways	128	32	74.4	38.8
SortedList.delete	114	19	54.7	26.2
SortedList.union	162	12	20.8	6.1
SortedList.diff	160	12	35.9	9.3
SortedList.insertionSort	149	11	24.2	4.8
StrictSortedList.insert	114	30	58.9	28.5
StrictSortedList.delete	114	19	53.9	26.2
StrictSortedList.union	162	12	19.7	6.2
UnaryNumerals.add	66	10	0.9	1.0
UnaryNumerals.distinct	91	4	0.8	0.7
UnaryNumerals.mult	66	11	17.0	5.1
BatchedQueue.enqueue	112	26	52.0	16.1
BatchedQueue.dequeue	88	35	26.7	14.5
AddressBook.makeAddressBook	63	33	24.8	6.6
AddressBook.merge	126	17	29.9	19.1
RunLength.encode	138	39	72.0	21.8
Diffs.diffs	83	24	51.4	11.7

Table 3.4 – Demonstrating the effect of indistinguishability heuristic. Benchmarks are run with PTE in automatic mode



Operation	Aspects off		Aspects on	
	#Programs		#Programs	
List.insert	101	1.4	99	1.5
List.delete	20772	16.3	6626	15.1
List.union	2527	5.2	837	5.6
List.diff	30821	13.3	12049	9.4
List.split	649	3.4	541	3.5
List.listOfSize	613	2.7	128	3.0
SortedList.insert	26837	24.3	9688	21.0
SortedList.insertAlways	26409	27.5	9249	25.3
SortedList.delete	22042	23.8	6952	20.4
SortedList.union	2431	6.8	716	5.9
SortedList.diff	13782	12.4	4914	8.1
SortedList.insertionSort	1399	3.7	296	3.0
StrictSortedList.insert	25358	30.9	9215	25.4
StrictSortedList.delete	21409	27.5	6762	24.2
StrictSortedList.union	2405	8.0	714	7.5
UnaryNumerals.add	193	6.0	182	5.7
UnaryNumerals.distinct	637	2.6	574	2.4
UnaryNumerals.mult	3757	8.3	2947	4.2
BatchedQueue.enqueue	16630	21.4	6191	16.7
BatchedQueue.dequeue	12990	24.5	6183	19.8
AddressBook.makeAddressBook	1001	7.6	999	8.6
AddressBook.merge	7591	21.2	5646	20.9
RunLength.encode	2092	33.8	602	28.1
Diffs.diffs	9641	11.5	3306	9.5

Table 3.5 – Demonstrating the effect of aspects. Benchmarks are run with STE in automatic mode

Operation	$c = 0$	$c = 2$	$c = 5$	$c = 10$	$c = 20$
List.delete	5.5	3.7	2.5	2.0	1.7
SortedList.insert	X	191.4	41.6	5.7	8.3
SortedList.insertAlways	X	202.9	48.4	6.7	8.0
SortedList.delete	8.6	8.1	7.6	7.0	6.6
StrictSortedList.insert	X	X	X	X	33.1
StrictSortedList.delete	X	4.1	2.1	1.8	1.6
BatchedQueue.dequeue	X	X	X	146.2	128.6
AddressBook.makeAddressBook	X	X	X	X	X
RunLength.encode	X	X	X	X	X

Table 3.6 – Demonstrating the effect of scores in the priority function. Benchmarks are run with PTE in manual mode

4 Program Repair

In previous chapters, we focused on synthesis, the process of generating code implementing a given specification. In this chapter, we change the problem setting: we assume that an implementation for the specification exists, but it is erroneous, in that it does not satisfy the specification for all inputs. Our task is then to identify which part of the implementation causes this inconsistency with the specification and then suggest a patch that, when replacing the erroneous code, will cause the specification to be satisfied for all inputs.

The key insight of our algorithm is that we can utilize the structure of the erroneous program as a guideline during repair. This follows from the hypothesis that a programmer often has correct insight when it comes to the high-level structure of the program, i.e., its control flow structure, but has made some small errors in specific program branches. Therefore, if the repair system can determine which branches cause the error, it can maintain the rest of the code and focus on generating localized fixes for the identified branches. This reduces the problem of repair from trying to implement the entirety of the function to generating potentially much smaller snippets of code. In that way, repair can scale to much larger programs than our synthesizer can handle.

After we localize the erroneous snippet, our repair system tries to generate a fix for it that satisfies the specification. To do so, we use a modified version of the synthesis techniques described in previous chapters that we call *similar term exploration*. Similar term exploration generates small variations to the original erroneous snippet, such as swapping operands of specific operators, adding the constant 1 to an integral expression, etc. This way, we add an additional level of reuse of the original code, which again contributes to the scalability of our approach to larger programs.

In the case that localization and/or similar term exploration fails, i.e., the entire program or the localized snippet, respectively, contains no useful structural information, we can always discard it and attempt synthesis from scratch. Synthesis can hence be viewed as a special case of repair, where the erroneous program is the empty program.

Part of this work was first presented in a publication by the author of this dissertation and others [KKK15]. This author's contributions were mainly the syntax and implementation of symbolic examples (discussed in Section 1.4.3), test minimization, the IF-CONDITION rule and the nondeterministic evaluator required to implement it, as well as implementation contributions throughout the system. In this dissertation, we present a more extensive evaluation of our technique, including the use of the new Probabilistic Term Enumeration (PTE) algorithm for test generation and similar term enumeration.

4.1 Example

As a running example, let us consider a function that computes the free variables of a term in a simple expression language. Variables are represented with an integer identifier. The function is specified with symbolic examples.

```
abstract class Expr
case class Var(i: BigInt) extends Expr
case class Unit() extends Expr
case class App(f: Expr, a: Expr) extends Expr
case class Lam(v: BigInt, b: Expr) extends Expr
case class Let(v: BigInt, vl: Expr, b: Expr) extends Expr

def fv(e: Expr): Set[BBigInt] = { e match {
  case Var(i) => Set(i)
  case Unit() => Set[BBigInt]()
  case App(f, a) => fv(f) ++ fv(a)
  case Lam(v, b) => fv(b) -- Set(v)
  case Let(v, vl, b) => fv(vl) -- Set(v) ++ fv(b) // Bug!
}} ensuring { res =>
  (e, res) passes {
    case Var(i) => Set(i)
    case App(Var(i), Var(j)) => Set(i, j)
    case Lam(BBigInt(0), Var(BBigInt(0))) => Set()
    case Lam(BBigInt(0), Var(BBigInt(1))) => Set(BBigInt(1))
    case Let(BBigInt(0), Var(BBigInt(0)), Var(BBigInt(1))) => Set(BBigInt(0), BBigInt(1))
    case Let(BBigInt(0), Var(BBigInt(1)), Var(BBigInt(0))) => Set(BBigInt(1))
    case Let(BBigInt(0), Var(BBigInt(1)), Var(BBigInt(2))) => Set(BBigInt(1), BBigInt(2)) } }
```

Function `fv` contains a bug: the variables `vl` and `b` have switched places. When invoked on this function, Leon first locates the source of the error on the right-hand side of the `Let` case, and then synthesizes the correct snippet to replace it:

```
e match {
  case Var(i) =>
```

```

    Set[BigInt](i)
  case Unit() =>
    Set[BigInt]()
  case App(f, a) =>
    fv(f) ++ fv(a)
  case Lam(v, b) =>
    fv(b) -- Set[BigInt](v)
  case Let(v, vl, b) =>
    (fv(b) -- Set[BigInt](v)) ++ fv(vl) // Fix: vl and b are now switched
}

```

Note how Leon narrowed down the source of the bug to a specific control flow branch of the program and left the correct portions of the program unmodified, thus reducing the effort needed to repair the program to a small fraction of what would be required to synthesize if from scratch. Additionally, Leon was able to synthesize a fix that is compatible with the specification.

4.2 Overview of the Algorithm

The repair process consists of four distinct stages:

- (1) *Test generation, classification and minimization.* We generate tests and classify them as passing or failing. If need be, we invoke the solver to ensure that we discover at least one failing test for each erroneous program. We then use a *test minimization* process that removes false negatives from our failing test suite.
- (2) *Fault localization.* We use the discovered failing tests and dedicated deductive repair rules to localize the source of the error.
- (3) *Synthesis of an alternative solution.* Having localized the error, we use a variant of synthesis to generate an alternative code snippet that satisfies the specification. This variant of synthesis tries to insert small modifications to the original erroneous snippet. If this approach fails, we fall back to regular synthesis, i.e., we discard the erroneous snippet and attempt to synthesize a correct one from scratch.
- (4) *Solution verification.* We attempt to verify the repaired program with the Leon verifier.

Our repair framework makes the assumption that the environment of the function implementation we are trying to repair is correct. Specifically, we assume that (1) the specification of the function under repair is correct, and (2) all functions transitively called by the function under repair and their specifications are also correct.

In the following sections, we analyze each stage of the repair process separately, we give a characterization of repairable programs, and we empirically evaluate our technique.

4.3 Test Generation

Tests are an important component of our fault localization procedure, since it uses a runtime approach based on concrete execution. Specifically, a branch of the program is characterized as erroneous if at least one test fails after entering that branch (and some other conditions hold, as discussed later). Therefore, it is important to generate tests that will cover every erroneous branch, and to take measures against wrong characterization of branches.

4.3.1 Sources of Tests

We use three sources of tests in our system.

User provided input-output examples. These are the most important tests, as they are part of the problem specification. In the case that these examples are symbolic, we generate a number of concrete tests for each of them by enumerating concrete values for each of their abstract patterns. For this we use the enumeration algorithm of Section 3.3 with the value grammar defined in Section 2.5.1.

For synthesis-by-example problems (where the specification consists only of concrete input-output examples), we disregard the next two sources of examples.

Value enumeration. We enumerate a large set of examples (currently set to 400) to have more chances to discover one that runs the erroneous branch. These examples are filtered by the precondition of the function. We use the same enumeration technique as in the previous paragraph.

Solver counterexamples. If the previous two processes fail to discover a failing test, we run the Leon verifier to discover a failing input. Of course, verification might succeed, in which case the need for repair is refuted.

4.3.2 Test Classification

We execute the initial code on the generated tests and classify them as passing or failing based on the result of the execution. A test is characterized as failing if, when executing the function under repair on it, a specification is violated or an error in the code is invoked. This may happen either immediately in the body of function under repair, or in any function transitively called by it. In the next section, we refine this initial classification to improve the characterisation of tests with regard to fault localization.

4.3.3 Test Minimization

Test classification as described above can misguide repair into overestimating the number of failing branches in the program. To illustrate this issue, let us consider another try to implement the example of Figure 4.1. This time, the error lies in the Var case.

Assume the tests collected are (a) `App(Var(0), Var(1))`, (b) `Lam(Var(0), Unit())`, (c) `Var(0)`, (d) `Var(1)` and (e) `Unit()`. When `fv` is ran on these tests, the call graph of Figure 4.1 is generated. A call to `fv` is represented by its argument. Solid borderlines stand for passing tests, dashed ones for failing ones.

Although there are technically four failing tests, it is obvious that the failures in the `App` and `Lam` tests should be attributed to the bug in the `Var` case. Generally said, inputs can be mistakenly falsified by errors in recursive invocations of the function under repair with different arguments.

To amend this, we run a *test minimization* process. As we run the function on the collected tests, we track the arguments of each invocation of the function under repair and generate the corresponding call graph. Tracked invocations include invocations on the collected test inputs, as well as recursive invocations on yet unknown inputs. Those newly discovered inputs are added to our collection of tests. We characterize all tests as passing and failing, and then *we discard each failing test that transitively invokes another failing test*.

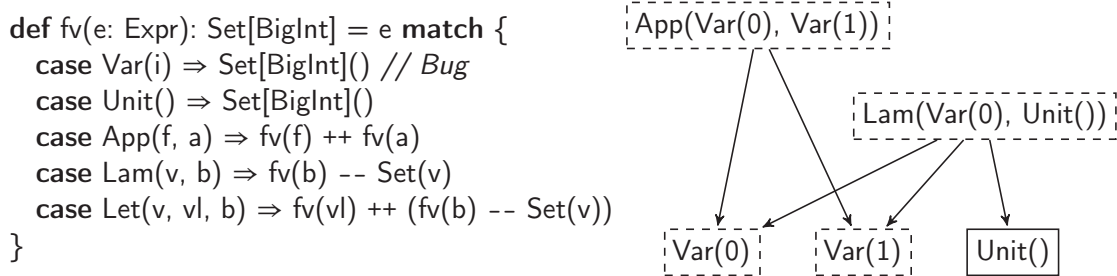


Figure 4.1 – Code and invocation graph for `fv`

QuickCheck [CH00, Cla12] is a testing tool for Haskell programs that also provides the option to minimize – or, in QuickCheck’s lingo, shrink – failing inputs. However, in QuickCheck the minimization process is based on the shape of the datatype and not the dynamic call graph generated by the input. For example, a failed input of a list type first gets shrunk to one of its sublists that also fails the specification; this process is repeated greedily (on the first failing sublist) until a list is found that does not fail the specification. For custom datatypes, the shrinking function has to be provided by the programmer, which limits the automation and generality of this approach.

4.4 Fault Localization

After we collect and minimize tests, we proceed to fault localization. By limiting the scope of the error, we can focus synthesis on a smaller part of the program and increase the size of programs we can repair. We follow a dynamic approach based on the collected failing tests. Since we are based on concrete inputs, this process is subject to failure as discussed in Section 4.7, but we found that this technique works reliably and efficiently in practice.

Like synthesis, fault localization successively applies a sequence of rules on the initial repair problem, taken from a set of deductive repair rules. This way, we use the same deductive synthesis framework for both synthesis and repair. These rules aim to narrow down the scope of the error to specific control flow branches. Note that this due to referential transparency of the purely functional PureScala, this process is sufficient to localize the error, as we do not have to account for mutable state possibly modified in another part of the program. Additionally, the minimization process of Section 4.3.3 ensures that we do not attribute to a branch of the program errors in other branches invoked in a recursive call.

During the localization process, we need to track the currently localized expression e , starting with the whole body of the function under repair. We do this with a witness in the path condition (see Section 1.3.1). We call this witness *guide* and write it as $\odot[e]$. We add $\odot[e_0]$ to the path condition of the initial repair problem, where e_0 is the whole body of the function under repair.

The fault localization rules used by our system are all normalizing deductive rules, which apply in priority to all other rules when the path condition contains a guide of a specific shape. Our fault localization rules are described next. We write \mathcal{F} for the minimized set of failing tests.

If-Focus. Given a problem $\llbracket \bar{a} \langle \odot[\text{if } (c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle x \rrbracket$, we first investigate if there exists another condition that fixes all the failing tests. Formally, we are trying to solve the formula

$$\exists C(\bar{a}). \forall \bar{i} \in \mathcal{F}. \phi[x \mapsto \text{if } (C) \{t\} \text{ else } \{e\}, \bar{a} \mapsto \bar{i}]$$

This higher-order condition is very hard to solve analytically, hence we follow a concrete approach: We substitute a nondeterministic boolean value $*$ for the initial condition c in the program and execute it with a nondeterministic evaluator described in Section 4.4.1. We focus on the condition if *at least one* path of the nondeterministic execution of *each* test succeeds.

This rule can be summarized as follows:

IF-FOCUS-CONDITION:

$$\frac{\begin{array}{c} \forall \bar{i} \in \mathcal{F}. \text{true} \in \phi[x \mapsto \text{if } (*) \{t\} \text{ else } \{e\}, c \mapsto *, \bar{a} \mapsto \bar{i}] \\ \llbracket \bar{a} \langle \odot[c] \wedge \Pi \triangleright \phi[x \mapsto \text{if } (x') \{t\} \text{ else } \{e\}] \rangle x' \rrbracket \vdash \langle P \mid T \rangle \end{array}}{\llbracket \bar{a} \langle \odot[\text{if } (c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid \text{if } (T) \{t\} \text{ else } \{e\} \rangle}$$

The substitution of c in the first line also applies to instances of c that are encountered in recursive calls. Also, observe that the nondeterministic execution of ϕ returns a set of boolean values as opposed to a single value. This set contains **true** if and only if at least one path of the nondeterministic execution succeeded.

If focusing on the if-condition fails, we try to focus on either branch. Here, things are simpler: if c evaluates to **true** for all failing tests, we will focus on the then-branch, and if it evaluates to **false** we will focus on the else-branch:

IF-FOCUS-THEN:

$$\frac{\begin{array}{c} \forall \bar{i} \in \mathcal{F}. c[\bar{a} \mapsto \bar{i}] \quad \llbracket \bar{a} \langle \odot[t] \wedge c \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle \end{array}}{\llbracket \bar{a} \langle \odot[\text{if } (c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid \text{if } (c) \{T\} \text{ else } \{e\} \rangle}$$

IF-FOCUS-ELSE:

$$\frac{\begin{array}{c} \forall \bar{i} \in \mathcal{F}. \neg c[\bar{a} \mapsto \bar{i}] \quad \llbracket \bar{a} \langle \odot[e] \wedge \neg c \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle \end{array}}{\llbracket \bar{a} \langle \odot[\text{if } (c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid \text{if } (c) \{t\} \text{ else } \{T\} \rangle}$$

If focusing on either branch also fails, we try to repair each branch separately and we reuse the if-condition.

IF-SPLIT:

$$\frac{\begin{array}{c} \llbracket \bar{a} \langle \odot[t] \wedge c \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P_1 \mid T_1 \rangle \quad \llbracket \bar{a} \langle \odot[e] \wedge \neg c \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P_2 \mid T_2 \rangle \end{array}}{\llbracket \bar{a} \langle \odot[\text{if } (c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle (c \wedge P_1) \vee (\neg c \wedge P_2) \mid \text{if } (c) \{T_1\} \text{ else } \{T_2\} \rangle}$$

Match-Focus. Similarly to **if**, we will use failing tests to focus on branches of a pattern match. Note that, since pattern matching can have more than two branches, we will not constrain MATCH-FOCUS on one branch, but rather all branches that are executed by at least one failing test. The rest of the branches are maintained unchanged. Our system currently cannot repair patterns, so there is no rule for **match** equivalent to IF-FOCUS-CONDITION.

In the following, c_i stands for the condition that the control flow reaches b_i , i.e., e matches the pattern C_i , but no previous C_j for $j < i$. Also, $v(C)$ stands for the variable bindings introduced by pattern C .

MATCH-FOCUS:

$$\frac{\begin{array}{l} B = \{ j \mid \exists \bar{i} \in \mathcal{F}. c_j[\bar{a} \mapsto \bar{i}] \} \quad \forall j \in B. [\llbracket \bar{a} \langle \odot[b_j] \wedge c_j \wedge \Pi \wedge v(C_j) \triangleright \phi \rangle x \rrbracket \vdash \langle P_j \mid T_j \rangle] \\ B' = \{ j \mid \neg \exists \bar{i} \in \mathcal{F}. c_j[\bar{a} \mapsto \bar{i}] \} \quad \forall j \in B'. [T_j \leftarrow b_j, P_j \leftarrow \text{true}] \end{array}}{\llbracket \bar{a} \langle \odot[\text{e match } \{ \dots \text{ case } C_j \Rightarrow b_j \dots \}] \triangleright \phi \rangle x \rrbracket \vdash \langle \bigvee (c_j \wedge P_j) \mid \text{e match } \{ \dots \text{ case } C_j \Rightarrow T_j \dots \} \rangle}$$

Val-Focus. When we encounter a local variable definition, we assume its value is correct and focus on the body of the definition. We also bind the defined variable in the path condition:

VAL-FOCUS:

$$\frac{\llbracket \bar{a} \langle \odot[b] \wedge \Pi \wedge v \leftarrow e \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid T \rangle}{\llbracket \bar{a} \langle \odot[\text{val } v = e; b] \wedge \Pi \triangleright \phi \rangle x \rrbracket \vdash \langle P \mid \text{val } v = e; T \rangle}$$

4.4.1 Nondeterministic Evaluator

The nondeterministic evaluator used for the IF-FOCUS-CONDITION rule is implemented on top of a regular evaluator for PureScala. Instead of returning a single PureScala value, it returns a stream of such values, computed as described next.

Most terminal operators do not introduce nondeterminism, and they return a singleton stream containing the value they return according to their deterministic semantics. The only operators that can introduce nondeterminism are the nondeterministic boolean value `*` introduced by the IF-FOCUS-CONDITION rule, and the **choose** operator. The semantics of the former is a stream consisting of the **false** and **true** values. For **choose**, the deterministic evaluator will invoke a satisfiability query on the Leon solver and return its first solution. The nondeterministic evaluator will generate a finite (lazy) stream of queries for the solver, each time adding the constraint that the new returned solution cannot be identical to any previous one.

For nonterminal operators, we compute the returned stream of values as follows: Let f_{op} be a metafunction describing the semantics of the n -ary operator op in terms of the semantics of its operands, i.e.,

$$\llbracket op(e_1, e_2, \dots, e_n) \rrbracket = f_{op}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \dots, \llbracket e_n \rrbracket),$$

then the semantics of the nondeterministic evaluator is given by the following stream comprehension:

$$\llbracket op(e_1, e_2, \dots, e_n) \rrbracket_{nd} = \{ f_{op}(s_1, s_2, \dots, s_n) \mid s_1 \leftarrow \llbracket e_1 \rrbracket_{nd}, s_2 \leftarrow \llbracket e_2 \rrbracket_{nd}, \dots, s_n \leftarrow \llbracket e_n \rrbracket_{nd} \}.$$

4.5 Similar Term Exploration

When the top level of a subproblem is not a control structure, fault localization is not able to focus further into one of its subterms. At this point, we invoke synthesis to search for an expression that will repair the error in the focused expression.

Despite not being able to localize the fault further, we still want to utilize the structure of the function under repair. Therefore, before invoking the full synthesis system, we invoke a term exploration rule that enumerates expressions similar to a given initial expression. We can use one of the term enumeration algorithms of Chapter 3, with a modified grammar that introduces small modifications to a guide expression.

4.5.1 Similar Term Grammar

Intuitively, the grammar of similar expressions for a guide $\odot[e]$ consists of small modifications to e , along with small productions taken from the general term grammar.

To define the similar term grammar, we begin with a regular grammar and attach to its starting symbol a new aspect S_e , where e is the given guide. Thus, $T_{\{S_e\}}$ is a nonterminal symbol whose productions will give us the terms of type T that are similar to e . Since it will always be $e : T$, we can omit the type of the nonterminal and write just S_e .

If $e = op(e_1, \dots, e_n)$, where op is the top-level operator of e , $S_{op(e_1, \dots, e_n)}$ can be decomposed as follows:

$$\begin{aligned}
 S_{op(e_1, \dots, e_n)} &::= \text{op}(\text{perm}^*(e_1, \dots, e_n)) \\
 &\quad | \quad \sim e \\
 &\quad | \quad \text{op}(S_{e_1}, e_2, \dots, e_n) \mid \dots \mid \text{op}(e_1, \dots, e_{n-1}, S_{e_n}) \\
 &\quad | \quad T_{\{+e\}\{D_2\}}
 \end{aligned}$$

Each component introduces a different modification to e :

- perm^* is a metafunction that computes the well-typed permutations of the arguments of op different than the original permutation. This will repair an expression whose arguments are in the wrong order. If op is a commutative binary operator, this component is omitted.
- $\sim e$ will give type-specific variations for e . These are
 - $e \pm 1$, if $e : \text{BigInt}$ or $e : \text{Int}$
 - $\neg e$, if $e : \text{Boolean}$
 - Instances of the other type constructors of the type of e , if op is a type constructor.
- $\text{op}(S_{e_1}, e_2, \dots, e_n)$ maintains all operands of op , except it substitutes e_1 by its similar productions. This rule explores the possibility that the top-level operator is correct and

the error lies deeper in the expression. It is a kind of fault localization within the similar term grammar.

- The final rule $T_{\{+e\}\{D_2\}}$ corresponds to the normal grammar productions for T , modified by two aspects $+e$ and D_2 .

$+e$ includes the guide e to the productions as a terminal rule. Including the guide allows the system to generate expressions with the guide as an operand. For instance, given guide e and a unary function f in scope, it will generate $f(e)$.

D_2 is called the *depth-bound* aspect, and bounds the produced expressions to a depth of 2. In general, D_k modifies production rules as follows:

- $T_{\{D_0\}} ::= \epsilon$,
- $T_{\{D_1\}}$ will maintain only the terminal rules of T and
- $T_{\{D_k\}}, k > 1$ will maintain all nonterminal rules for T , except the aspect D_{k-1} will be attached to all operands. I.e., $T ::= \text{op}(T_1, T_2, \dots, T_n)$ becomes $T_{\{D_k\}} ::= \text{op}(T_{\{D_{k-1}\}}, T_{\{D_{k-1}\}}, \dots, T_{\{D_{k-1}\}})$

If e is an expression that cannot be decomposed, i.e., a nullary operator, the first and third components of S_e are omitted.

If we want to generate a probabilistic similar terms grammar, we attach uniform probabilities to the rules generated for each nonterminal.

4.6 Verification

Finally, we try to verify the correctness of the function with our suggested fix by deploying the Leon verifier. This is because the correctness of fault localization depends on the completeness of code coverage of our selected examples, hence we might have missed some erroneous branches during repair. When we cannot verify (or disprove) the correctness of the solution at this step, the solution will be reported to the user as unverified, and manual inspection is required to ensure that the suggested solution implements the intention of the programmer. This is an innate limitation of any system that handles a Turing-complete language. We usually find that the generated solutions for our benchmarks are the desired ones, since most of those benchmarks are written with strong specifications.

4.7 Characterization of Repairable Programs

For a program to be successfully repaired by our algorithm, the following conditions have to be met:

- Collected tests cover all erroneous control-flow branches of the program. This is necessary since we need a failing execution for each error to localize it. With the different

sources of tests, we found this is usually achieved for our benchmarks.

- Test minimization has to maintain at least one failing test for each bug in the function. Recall that a failing test is maintained by minimization if all recursive calls during its execution are passing tests. Therefore, if *each* failing test corresponding to a bug in the program invokes another failing test, the minimized test suite will not uncover this bug and will be missed by fault localization.

To demonstrate this effect, consider the following —admittedly artificial— program that tries to compute the size of a list:

```
def size[A](l: List[A]): BigInt = l match {
  case Nil() => -1
  case Cons(_, t) => size(t) - 1
} ensuring { res => res ≥ 0 }
```

This program contains bugs in both the Nil and Cons branches. During minimization, every Cons failing test will be found to invoke another failing test: if it has only one element, it will invoke the function with the Nil argument, and if it has more than one element, it will invoke it with a smaller Cons. Therefore, the minimized test suite will contain only the Nil testcase, and fault localization will fail by localizing only on the Nil branch. This will result in only the Nil branch being repaired.

This particular problem could be amended by invoking repair again on the partially repaired program after it fails the final verification phase of our system; however, our system is not currently configured to do that.

Given a higher number of tests and a more complicated program, it is unlikely that a bug will coincidentally always invoke failing tests, and we did not observe this behavior in our benchmarks.

- Fault localization has to successfully localize the bugs within the control structure of the program. A case that would not be handled properly is an if expression whose condition and at least one branch is wrong. Another, more realistic case, is a local variable definition containing a bug in the value assigned to the variable.
- Synthesis has to discover a suitable fix. If the bug is small, as per our working hypothesis, and is contained within the space of programs generated by the similar term grammar, then the synthesis will always discover it. Otherwise, the discovery of the fix is subject to the limitations of normal synthesis.

4.8 Evaluation

We evaluate our techniques on a set of functional programs, to each of which we manually inserted a set of errors. Most of these programs manipulate data structures.

First, we developed a correct version of each benchmark, including all necessary data structures and function definitions, and verified it with using Leon. Then, we generated a number of copies of each benchmark and, for each of those copies, we picked a function definition and manually inserted one or more errors in it. Some errors are taken from the error model implied by the similar term grammar, while others are too extensive to be repairable by similar term exploration. In the latter case, the system has to fall back to normal synthesis. When running the benchmarks, we give as input to Leon the name of the function that we want to repair. This is to avoid wasting time trying to verify every function in the program.

The evaluation results are presented in tables 4.1 and 4.2. For each benchmark, we include:

- the name of the benchmark and the function to be repaired,
- a description of the error. Errors types include: a small variation to a branch of the program, a completely erroneous match-case, and two separate small variations in different branches of the function.
- as an indication of the size of the benchmark, the size in AST nodes of each of (1) the complete program, (2) the function under repair, (3) the localized erroneous branch and (4) the fix generated by STE.
- the time spent in test collection, classification and minimization,
- For a number of configurations of Leon, the time spent for repair (including fault localization), as well as whether or not Leon was able to prove the correctness of the generated repair. Note that a non-verifiable repair might still capture the intention of the programmer, and we determined that this is often the case after manual inspection of the results.

We can use different term grammars for repair. The chosen grammar is the basis for the similar term grammar as explained in Section 4.5.1, and is used in full synthesis when similar term exploration fails. We evaluated the benchmarks in the following configurations:

- using Symbolic Term Exploration with built-in grammars (STE).
- using Probabilistic Term Enumeration with built-in grammars (P0). Recall that built-in grammars are not intrinsically probabilistic and are assigned uniform probabilities for all rules.
- using Probabilistic Term Enumeration with built-in grammars as before, but with similar term exploration deactivated (P0-NoS).
- using Probabilistic Term Enumeration with the plain probabilistic grammar extracted from a corpus of Leon benchmarks described in Section 2.7 (P1), and

- using Probabilistic Term Enumeration with the annotated probabilistic grammar described in the same section (P2).

For custom grammars, we include an additional grammar file extracted from the specific benchmark. This serves two purposes: to include productions for locally defined types that do not exist in the general grammar file, and to bias the grammar towards the structures utilized more often in the benchmark. To compute the grammar probabilities, we take all grammar files relevant for a benchmark, process them, and then compute probabilities from the integer measures in the resulting grammar rules, as in Section 2.6. We do not use any weighting between the general and benchmark-specific grammar files; this is an approach on which we could improve.

We use aspect grammars for all but the last configuration. We did not use indistinguishability for any configuration, because repairs are usually smaller in scale than full synthesis and therefore the overhead of indistinguishability would not be justified. Also, we do not use the separate recursive call rule, but generate safe recursive calls within term exploration. All benchmarks and the introduced errors can be found in Appendix C.

Assessment of results. As we can see in Table 4.1, fault localization is remarkably efficient, and usually manages to localize the error in a small subset of the function under repair. Test collection is also efficient, taking only nine seconds at the longest, and definitely justified as a means to improve the efficiency of repair.

Consider Table 4.2, presenting the results of running repair on each benchmark. Leon successfully handles a number of benchmarks, including tougher ones with more than one error or extensive errors. The repair times are generally short and only increase significantly when the error is extensive enough that Similar Term Exploration cannot come up with a modification that will fix the code, after which we resort to full synthesis. One of these hard benchmarks is the List.count benchmark of Appendix C.3, with the Cons branch being completely wrong:

```
def count(e: T): BigInt = { this match {
  case Cons(h, t) =>
    if (h == e) { // This branch body replaced with BigInt(0) in benchmark
      1 + t.count(e)
    } else {
      t.count(e)
    }
  case Nil() =>
    0
}} ensuring { res =>
  res + (this - e).size == this.size
}
```

Operation	Error	Size				Test ⌚
		Prg	Fun	Err	Fix	
Compiler.desugar1	full case	676	82	3	5	1.9
Compiler.desugar2	full case	674	80	2	6	1.9
Compiler.desugar3	variation	678	84	7	7	1.5
Compiler.desugar4	variation	678	84	7	7	1.9
Compiler.desugar5	2 variations	678	84	14	14	1.9
Compiler.simplify	variation	724	31	4	4	1.1
Compiler.semUntyped	full case	671	78	1	4	1.1
Heap.merge1	variation	347	37	3	9	3.9
Heap.merge2	variation	347	37	1	1	3.8
Heap.merge3	variation	347	37	3	9	3.8
Heap.merge4	variation	347	37	9	15	3.4
Heap.merge5	variation	349	39	5	9	3.5
Heap.merge6	2 variations	347	37	2	2	4.1
Heap.insert	variation	310	8	8	10	9.0
Heap.makeNode	variation	349	16	7	5	3.7
List.pad	variation	808	35	8	6	1.2
List.++	variation	718	10	3	5	1.5
List.:+	variation	750	12	1	3	1.2
List.replace	variation	752	22	6	6	1.2
List.count	variation	805	10	3	12	1.4
List.find1	variation	805	23	2	4	1.2
List.find2	variation	807	25	4	6	1.2
List.find3	variation	808	25	4	4	1.2
List.size	variation	755	11	4	4	1.5
List.sum	variation	752	11	4	4	1.8
List.-	variation	752	17	1	3	2.4
List.drop	variation	793	25	4	4	1.4
List.drop	variation	793	23	3	5	2.0
List.&	full case	752	18	4	5	1.5
List.count	variation	752	8	1	12	1.4
Numerical.power	variation	178	23	5	7	0.9
Numerical.moddiv	variation	127	21	3	3	0.7
MergeSort.split	full case	233	22	5	7	2.9
MergeSort.merge1	variation	235	33	7	11	3.9
MergeSort.merge2	variation	235	33	3	7	4.3
MergeSort.merge3	variation	233	31	5	11	3.6
MergeSort.merge4	variation	235	33	1	1	3.1

Table 4.1 – Benchmarks repaired by Leon: error types, sizes, and test generation times






Operation	STE		P0		P0-NoS		P1		P2	
		⊢		⊢		⊢		⊢		⊢
Compiler.desugar1	3.6	✓	2.1	✓	2.6	✓	2.2	✓	2.7	✓
Compiler.desugar2	5.3	?	4.5	✓	3.2	?	5.7	✓	3.4	✓
Compiler.desugar3	3.9	✓	2.0	✓	3.1	✓	2.1	✓	2.2	?
Compiler.desugar4	4.0	✓	3.1	✓	3.2	✓	3.4	✓	3.2	✓
Compiler.desugar5	6.3	✓	3.9	?	3.9	✓	3.7	?	4.2	✓
Compiler.simplify	3.4	✓	1.9	✓	2.5	✓	1.8	✓	2.8	✓
Compiler.semUntyped	19.4	✓	23.2	✓	28.7	?	X	X	5.9	✓
Heap.merge1	6.1	✓	5.3	✓	X	X	5.8	✓	5.3	✓
Heap.merge2	3.3	✓	2.4	✓	2.1	✓	2.1	✓	2.8	✓
Heap.merge3	6.2	✓	5.4	✓	X	X	5.7	✓	5.7	✓
Heap.merge4	5.0	✓	3.9	✓	X	X	5.2	✓	4.1	✓
Heap.merge5	5.8	✓	7.7	✓	9.7	✓	6.1	✓	16.6	✓
Heap.merge6	4.5	✓	2.8	✓	2.8	✓	3.1	✓	3.2	✓
Heap.insert	3.6	✓	2.9	✓	4.5	✓	3.1	✓	3.1	✓
Heap.makeNode	6.8	✓	8.3	✓	5.6	✓	9.5	✓	8.6	✓
List.pad	2.6	✓	1.8	✓	3.7	✓	1.5	✓	2.5	✓
List.++	2.8	✓	1.5	✓	3.1	✓	1.4	✓	9.4	✓
List.:+	3.6	✓	1.0	✓	2.0	✓	1.0	✓	2.7	✓
List.replace	4.5	✓	1.8	✓	4.7	✓	1.8	✓	2.1	✓
List.count	2.4	✓	1.3	✓	1.6	✓	1.3	✓	1.5	✓
List.find1	2.7	✓	1.8	✓	4.3	✓	1.9	✓	1.9	✓
List.find2	3.2	✓	1.9	✓	4.6	✓	1.8	✓	1.9	✓
List.find3	3.3	✓	1.8	✓	3.3	✓	1.8	✓	1.6	✓
List.size	2.5	✓	1.1	✓	2.2	✓	1.1	✓	1.4	✓
List.sum	2.9	✓	1.3	✓	2.7	✓	1.3	✓	1.6	✓
List.-	1.8	✓	0.6	✓	2.3	✓	0.7	✓	2.5	✓
List.drop	2.9	✓	1.5	✓	3.8	✓	1.6	✓	1.4	✓
List.drop	4.5	✓	3.5	✓	3.3	✓	30.0	✓	3.0	✓
List.&	3.1	✓	2.4	✓	3.5	✓	3.3	✓	59.7	✓
List.count	51.2	✓	10.6	✓	12.3	✓	178.3	✓	X	X
Numerical.power	3.4	✓	1.3	✓	X	X	1.4	✓	X	X
Numerical.moddiv	2.1	✓	1.0	✓	1.3	✓	0.8	✓	1.4	✓
MergeSort.split	3.7	✓	2.7	✓	9.5	✓	2.8	✓	9.4	✓
MergeSort.merge1	3.2	✓	2.6	✓	7.7	✓	2.5	✓	2.8	✓
MergeSort.merge2	6.4	✓	5.6	✓	5.2	✓	5.5	✓	6.1	✓
MergeSort.merge3	3.6	✓	2.5	✓	6.5	✓	2.4	✓	X	X
MergeSort.merge4	2.6	✓	1.8	✓	1.7	✓	1.8	✓	1.9	✓

Table 4.2 – Benchmarks repaired by Leon: localization and repair times

Similar Term Exploration and Probabilistic Term Enumeration perform similarly with the built-in grammar, with the latter having a significant advantage in the most challenging benchmark (about 50 to 10 seconds). Unfortunately, the custom grammars do not perform well. One reason for that might be that different components of the grammars do not mesh well: the general grammar, the grammar from the repair benchmark, and the grammar of recursive functions might be weighted in a manner that does not correspond to a reasonable distribution of expressions.

Finally, the usefulness of similar term exploration is displayed when inspecting the second and third configurations of Table 4.2, which are identical, except similar term exploration has been deactivated for the latter. Benchmarks are generally slower, with 4 benchmarks being unrepairable, and 2 additional ones generating repairs that cannot be verified by the Leon verifier.

4.9 Conclusion

In this chapter, we presented a system capable of localizing and repairing bugs in functional programs. The algorithm collects tests from different sources and classifies them as passing or failing. Failing tests are filtered by a trace minimization algorithm to reject those that can be attributed to the failure of another test. Fault localization uses the remaining failing tests to localize the error within the control flow branches of the program. Then, a modified synthesis algorithm suggests fixes for the discovered bugs. This algorithm uses a grammar that generates terms similar to the original erroneous terms, and falls back to normal synthesis if this grammar yields no solution. Finally, we characterize the programs repairable by our algorithm and experimentally evaluate our technique.

5 Related Work

5.1 Synthesis Systems

Program synthesis has been a challenge in computer science since its early days (see, for example, [MW71]). The seminal SKETCH project [SLTB⁺06] was the first to demonstrate the feasibility of using modern constraint solving technology in synthesis tools, and used the CEGIS framework to solve synthesis problems. This spawned a sizable body of research on synthesis, materializing in a number of synthesis systems, each of which focuses on specific aspects and forms of the problem.

SYNTREC [IQLS15] and SYNAPSE [BTGC16] use a similar approach based on user-defined *generators* (or metasketches) that describe high-level, reusable patterns of computation, in the spirit of SKETCH. The programmer interacts with the system by providing an appropriate generator for the task at hand, which is then used by the system to synthesize a complete program. SYNTREC validates candidate program with bounded checking, whereas SYNAPSE uses SMT solving. These approaches scale better than Leon for some benchmarks, but require the programmer to have significant insight into the form of the resulting program.

In SYNQUID [PKS16], the target specification is given in the form of a liquid type [RKJ08]. Additionally, the user provides the set of usable program components. The authors modify the liquid type inference algorithm to enable top-down breakdown of a liquid type, and use the inference rules as deductive synthesis rules. Conditionals are generated with a form of condition abduction. Compared to Leon, SYNQUID specifications tend to be much longer and require more insight, as the programmer needs to provide the liquid type signatures of all intermediate components used by the synthesizer.

MYTH [OZ15] is a tool that synthesizes programs with higher-order functions, ADT applications and pattern matching. The specification is given in the form of input-output examples. Similarly to Leon, pattern matching is handled outside the main enumeration procedure. To ensure that no redundant terms are generated, MYTH generates terms in β -normal, η -long form. The enumeration itself uses breadth-first search and uses *refinement trees* to represent

the generated programs, a structure very similar to our own derivation trees. Compared to our tool, MYTH cannot handle formal specifications, and the authors do not present benchmarks with arithmetic operations other than index increment and decrements for lists.

λ^2 [FCD15] is another tool which performs inductive synthesis, focusing on higher-order functions. Similarly to us, it maintains a queue of partial expressions that in λ^2 are called *hypotheses*. These are gradually refined by substituting other hypotheses in place of their holes. Open hypotheses, i.e., invocations of higher-order functions, are chosen from a fixed list, while closed ones are generated by bottom-up enumeration. λ^2 uses domain knowledge of the behavior of higher-order combinators to push examples into the body of the combinator. While doing so, it can filter out hypotheses that are inconsistent with those examples. This corresponds to the optimization of Section 3.3.5, but in λ^2 it does not apply for first-order terms.

ESHER [AGK13] and LASy [PGGP14] use a set of input/output examples and a set of program components to automatically synthesize progressively more complicated code snippets, until one is discovered which satisfies all input-output pairs.

Another approach [FOWZ16] views an input/output example as a singleton refinement type. A solution is satisfactory if its type is a supertype of all provided examples.

FLASHMETA [PG15] is a generic framework for synthesis-by-example. The framework provides a fixed synthesis algorithm and can be instantiated with a specific DSL, along with weights for its expressions and other domain-specific knowledge. The synthesizer is in dialogue with the programmer to eliminate ambiguities in the generated programs. FLASHMETA uses version space algebras (VSAs) [Mit81] to compactly represent spaces of programs. VSAs support efficient operations on program spaces, such as union, intersection and ranking.

DACE [WDS17] synthesizes spreadsheet programs with an emphasis on extracting information from ranges of cells. First, it learns a finite tree automaton (FTA) that represents a program compatible with an input-output example. It then combines as many of those automata as possible with FTA intersection, and introduces conditionals to combine the intersection groups. BLAZE [WDS18] implements follow-up work based on these ideas. BLAZE constructs an *abstract finite tree automaton* (AFTA), a structure similar to finite tree automaton which operates on an abstract semantics of a provided DSL. Since the abstract semantics overapproximate the concrete semantics, some programs permitted by the AFTA might fail on some examples. BLAZE uses such programs to refine the AFTA. The authors use their technique to implement programs on string, matrix and tensor transformations.

NEO [FMBD18] is a tool that uses *equivalence module conflicts* to speed up synthesis. Given a wrong partial program P , it creates an SMT formula corresponding to the specifications of all nodes of P , and discovers a minimal unsatisfiable core for this SMT formula. Components whose specification implies a clause of this core will falsify the specification. They are assigned to the same equivalence class for this conflict and not considered by synthesis.

Perelman et al. [PGGP14] represent program spaces with custom DSLs given as context-free grammars. Their system syntactically checks generated terms for redundancy based on arithmetic operator laws and reject redundant terms. Contrary to their technique, our disambiguation technique based on aspect grammars operates at the grammar level, and so redundant terms are never generated in the first place.

Syntax-guided synthesis (SyGuS) has emerged as a common formulation and interchange format in which to express many program synthesis problems [ABJ⁺13]. SyGuS format's input to the synthesis problem is given as a context-free grammar along with a specification in a specified background theory. EuSolver [ARU17] and the CVC4 SMT-solver enhanced with synthesis procedures [RKK17, RDK⁺15, RKT⁺17] are two prominent solvers in the SyGuS domain. We would like to highlight that program synthesis and code repair within Leon is a much more challenging problem than synthesis within SyGuS and related systems: the main difficulties include the rich type system and algebraic data types of Scala, and the synthesis of recursive functions.

The TRANSIT system [URD⁺13] and ESolver [ABJ⁺13] are some of the first system to enumerate pairwise distinguishable expressions in the realm of synthesis. In contrast to Leon, those tools enumerate expressions bottom-up. The SyGuS solver EUSolver combines indistinguishability-based enumeration with decision-tree based condition inference [ARU17]. As mentioned before, none of these tools can handle recursive functions or ADTs.

COZY [LTE16] synthesizes data structures from specifications of element retrieval operations. First, it discovers an outline of the operations evoked to implement each retrieval query using a variant of CEGIS. The discovered solutions are ranked based on their estimated performance. Then, a suitable data structure representation is chosen for the discovered outline. The authors use their synthesized data structures to successfully replace hand-written data structures from a range of real-world applications. In a recent update [LET18], COZY was updated to handle a wider range of specifications and imperative updates to the data structures. Starting from a trivial implementation of queries and an abstract representation of data, it uses an iterative process of concretizing the used data structures and improving synthesized queries. Synthesized updates to data structures are incremental to improve the synthesized code's performance. Although COZY and Leon overlap in some of their underlying techniques, COZY focuses on a narrower application domain and produces more complex programs within this domain. In contrast, Leon focuses on more fundamental synthesis techniques and can handle a wider range of problems, including recursion, arithmetic computations, and synthesis by example.

Finally, a direction of work has been synthesizing snippets that interact with APIs. Since large APIs are an integral part of programming, the focus of this work is shifted to higher-level code that is mostly restricted to a series of API calls as opposed to application of primitive operations. These tools usually require a corpus of code in the target language to construct a language model offline, from which they extract weights which guide the synthesis algorithm. Reinking

and Piskac [RP15] focus on repair of type-incorrect API invocations. The line of work of Gvero et al. [GKP11, GK15, GKGP13] aims to synthesize queries to APIs in Scala/Java within an IDE environment, using the local environment at the point of invocation of the tool (including local variables and API functions), or, more recently, taking a free form query as input. In [PGBG12], the input to the synthesizer is a partial expression, which can encode calls to an unknown function on known arguments or, given an object, an invocation of an unknown method or lookup of an unknown field of that object. A synthesis algorithm completes those partial expressions to obtain a complete program.

5.2 Program Space Representations

Both Perelman et al. [PGGP14] and the SyGuS format [ABJ⁺13] use context-free grammars to represent program spaces.

Previous versions of Leon [KKK15] represent a set of programs corresponding to a grammar with a composite expression with boolean guards; each assignment of values to these guards represents a program from this set.

Probabilistic context-free grammars (PCFGs) are a classical extension of context-free grammars [JM08, Ch. 14]. They may be used both to model ambiguity (for applications in natural language processing), and to model probability distributions over the generated language, which motivates our application in accelerating code synthesis. The more recent model of probabilistic higher-order grammars (PHOGs) [BRV16, RBV16] extends PCFGs by allowing the expansion probabilities of a non-terminal node to depend on attributes such as node siblings and DFS-predecessors. Experiments indicate that the PHOG model is significantly better at predicting elements of JavaScript programs than PCFGs. Extending the probabilistic model of our dissertation to use PHOGs instead of PCFGs is an area of future work.

With increasing availability of large open-source code repositories such as GitHub and Bitbucket, the statistical analysis of code corpora has become an exciting research problem. Code repositories have been used to learn coding idioms [AS14], to automatically suggest names for program elements [ABBS15], and to deobfuscate JavaScript code [RVK15].

The program completion tool Slang [RVY14] uses n -grams and recurrent neural networks to predict missing API calls in code snippets. There are two main aspects which distinguish our work from Slang: (1) the presence of hard correctness requirements in Leon in the form of pre-/post-conditions, and (2) program synthesis in Leon is fundamentally about synthesizing expressions rather than API call sequences, and prediction systems such as n -grams are insufficient to create the nested recursive structure inherent in the output we produce.

The DeepCoder tool [BGB⁺17] uses a recurrent neural network (RNN) to predict the presence of elements in the program being synthesized. The output of this neural network is then used to guide a more exhaustive search over the space of possible programs.

In recent work, Brockschmidt et al. [BAGP18] generate a missing line of imperative code from a context consisting of the code around the line, as well as accessible variables. Similar to us, they represent a partial synthesis solution as an incomplete AST. Every missing node in the partial AST is assigned a probability that is a function of the parent and sibling nodes of the node, uses of available variables in the partial solution, neighboring tokens in the resulting code, etc. They model these probabilities with a neural network trained on a corpus of code. Compared to our probabilistic grammars, this is a much more elaborate modelling of the target language. However, their work does not deal with any form of specification to verify the correctness of their synthesized snippets, and the first suggested snippet is often not the desired one. The generated snippets are much smaller compared to Leon. Finally, a drawback of this approach is the need of substantial additional code around the point where synthesis is invoked.

5.3 Repair

Related work on repair for functional programs is scarce. Previous work has mostly focused on imperative programs, without access to built-in ADTs and with greater focus on handling of statements with side-effects.

GenProg [GNFW12] and SemFix [NQRC13] operate on C programs, enhanced with sets of passing and failing test cases. Additional specifications in the form of assertions or designated outputs for specific inputs are not present. Since these systems operate on imperative programs with side-effects, our fault localization technique is not applicable. Instead, those tools employ dynamic statistical fault localization techniques. The repair strategy varies between those tools: on the one hand, GenProg synthesizes no new statements, and instead tries to generate repairs by swapping, deleting, or duplicating existing program statements using a genetic algorithm. On the other hand, SemFix deploys synthesis, but unlike Leon does not use existing code as a guide.

AutoFix-E/E2 [PWF⁺11, PFNM15, WPF⁺10] attempts to repair Eiffel programs with formal specifications. However, those contracts are only used to automatically generate and classify test cases, and are not used to verify repairs. AutoFix-E uses a complex fault localization mechanism, combining syntactic, control flow and statistical dynamic information. Its repair mechanism is based on built-in repair schemas, which allow reuse of the faulty statement as a component of the repaired expression.

Samanta et al. [SDE08] repair sequential boolean programs. They compute Hoare triples representing each boolean statement in the program, then check them for correctness on a specific order and proceed to repair those which are found to be erroneous. In later work, they proceed to repair C programs [SOE14] by abstracting them with boolean constraints. This constraint is then repeatedly modified according to a set of update schemas until all assertions in the program are satisfied, at which point the boolean constraint back to a repaired C program. The update schemas are picked based on a cost model. Their approach

needs developer intervention to define the cost model for each program, as well as at the concretization step.

SPR [LR15] is a system that repairs imperative programs from input-output examples. It uses a fixed set of repair schemas with abstract values, and discovers parameters for those sketches so the program passes the test cases. It also synthesizes conditional expressions which partition the space of inputs each sketch needs to handle.

Logozzo et al. [LB12] present a framework which suggests repairs, taking input from the static checker CodeContracts [FL11]. Suggested repairs address errors that are discovered by common static analyses, such as arithmetic overflows or object initialization, and are typically simpler than those Leon can generate.

Gopinath et al. [GMK11] use concrete values and a SAT solver to generate repairs for data structure operations. First, they pick a concrete input which exposes a suspicious statement, then they use a SAT-solver to discover a corresponding concrete output that satisfies the specification. This concrete output is then abstracted to various possible candidate expressions, which are filtered with bounded verification.

Another approach to fault localization is suggested by Chandra et al. [CTBB11]: they examine an expression as a possible error source if replacing it with a ground term fixes a failing execution.

Repair has also been studied in the context of reactive and pushdown systems with otherwise finite control [JGB05, JSGB12, vEJ13, GBC06]. In other work [vEJ13], the authors discuss how their repairs preserve specific traces of the original program. Our own repairs reuse parts of the original programs and thus have a tendency to preserve correct traces, but an analysis of semantic guarantees is left for future work.

An important component of our repair system is test generation. We use our enumeration algorithms on term grammars to generate test cases. A more elaborate approach is taken by the Korat [BKM02] automatic test generation system, which generates inputs for Java data structure programs, exhaustively up to a specific size. To reduce the search space of inputs, Korat generates only nonisomorphic inputs, according to a definition of isomorphism. Inputs are filtered by the class invariant of the data structure.

6 Conclusions and Future Work

In this dissertation, we presented techniques to improve the efficiency of synthesis and repair of recursive functional programs.

We gave a brief overview of the Leon deductive synthesis tool and described how we updated its deductive synthesis rules to introduce specific safe recursive calls in a problem's path conditions and handle bound variables.

We gave an overview of different flavors of term grammars used to describe program spaces in Leon. We presented a version of aspect grammars for synthesis which remain context-free for a specific choice of aspects. An interesting future direction would be to incorporate aspect grammars into more powerful formalisms, such as probabilistic higher-order grammars or version-space algebras. Additionally, we presented generic grammars, whose rules contain type parameters which can be instantiated to types appearing in the program to generate ground productions. The way we instantiate generic types for probabilistic grammars could be improved so that the final grammars contain a more accurate proportion of instantiated rules and rules that were originally ground. We described the built-in grammar of Leon, as well as a system to manually define term grammars, or extract them by analyzing a corpus of code. As future work, we would like to increase the expressiveness of extracted probabilities by making them conditional on additional features other than the parent node in the AST, for example, sibling nodes, number of parameters of the function under synthesis, and contextual information such as what recursion schema is being used. Additionally, given a specific synthesis problem, we would like to compute posterior probabilities based on the priors coming from the language model and the shape of the problem's specification. Neural networks can be trained to approximate these prior and posterior probabilities.

Next, we described two different enumeration algorithms which are deployed to discover solutions to a synthesis problem. We show that those algorithms solve a variety of benchmarks of the realm of recursive functional programs. Additionally, we demonstrate that the optimizations we suggest to improve the efficiency of synthesis do have practical impact on our benchmarks. We recently deployed similar ideas to efficiently generate terms in the

realm of relational algebra [WSK⁺18]. As a future direction, we would like to investigate the possibility to subsume the full Leon deductive synthesis framework with the Probabilistic Term Enumeration algorithm. Deductive rules could be encoded as grammar rules and a unified probabilistic model could be created for all expressions, including **match**-expressions and recursive calls, which are currently handled by dedicated deductive rules. This would require an extension of the grammars to handle, for instance, newly bound variables that are currently introduced by those rules. We believe this is achievable within the aspect framework.

Finally, we presented a system which is able to localize and fix bugs in functional programs. Fault localization is based on execution traces, and the system deploys a novel trace minimization technique to prune false positives. Since our technique depends on code coverage of a test suite, in the future we would like to utilize techniques that improve the code coverage of test suites. Fixes for the discovered bugs are generated with a variant of synthesis that generates terms similar to the original erroneous term. We showed that our technique is able to localize and fix a variety of bugs in functional programs.

In conclusion, we believe that the work presented in this dissertation improves the state of the art in the field of functional synthesis and repair, and can be used to improve the performance of existing synthesis techniques, as well as a starting point for future work in the field.

A Synthesis Benchmarks by Specification

We present the synthesis by specification benchmarks used in Chapter 3. For each benchmark, we give the code containing the synthesis problem (the **choose** construct, or the synthesis hole `???`), followed by the synthesized solution.

A.1 List.insert

```
def insert[A](in1: List[A], v: A) = choose {  
  (out : List[A]) =>  
    out.content == in1.content ++ Set(v) }
```

// Solution:

```
def insert[A](in1 : List[A], v : A): List[A] = {  
  Cons[A](v, in1)  
} ensuring {  
  (out : List[A]) => out.content == in1.content ++ Set[A](v) }
```

A.2 List.delete

```
def delete(in: List[BigInt], v: BigInt) = ???[List[BigInt]] ensuring {  
  (out : List[BigInt]) =>  
    out.content == in.content -- Set(v) }
```

// Solution:

```
def delete(in : List[BigInt], v : BigInt): List[BigInt] = {  
  in match {  
    case Nil() => List[BigInt]()  
    case Cons(h, t) =>  
      val rec = delete(t, v)  
      if (h == v) {
```

```

    rec
  } else {
    Cons[BigInt](h, rec) } }
} ensuring {
  (out : List[BigInt]) ⇒ out.content == in.content -- Set[BigInt](v) }

```

A.3 List.union

```

def union[A](in1: List[A], in2: List[A]) = choose {
  (out : List[A]) ⇒
    out.content == in1.content ++ in2.content }

```

// Solution:

```

def union[A](in1 : List[A], in2 : List[A]): List[A] = {
  in1 match {
    case Nil() ⇒ in2
    case Cons(h, t) ⇒ Cons[A](h, union[A](t, in2)) }
} ensuring {
  (out : List[A]) ⇒ out.content == in1.content ++ in2.content }

```

A.4 List.diff

```

def delete[A](in1: List[A], v: A): List[A] = {
  in1 match {
    case Cons(h,t) ⇒
      if (h == v) delete(t, v)
      else Cons(h, delete(t, v))
    case Nil() ⇒ Nil[A]() }
} ensuring { _ .content == in1.content -- Set(v) }

```

```

def diff[A](in1: List[A], in2: List[A]) = choose {
  (out : List[A]) ⇒
    out.content == in1.content -- in2.content }

```

// Solution:

```

def diff[A](in1 : List[A], in2 : List[A]): List[A] = {
  in2 match {
    case Nil() ⇒ in1
    case Cons(h, t) ⇒ delete[A](diff[A](in1, t), h) }
} ensuring {
  (out : List[A]) ⇒ out.content == in1.content -- in2.content }

```

A.5 List.split

```
def splitSpec[A](list : List[A], res : (List[A],List[A])) : Boolean = {
  val (l1, l2) = res
  val s1 = l1.size
  val s2 = l2.size
  abs(s1 - s2) ≤ 1 && s1 + s2 == list.size &&
  l1.content ++ l2.content == list.content }
```

```
def abs(i : BigInt) : BigInt = {
  if(i < 0) -i else i
} ensuring(_ ≥ 0)
```

```
def split[A](list : List[A]) : (List[A],List[A]) = {
  choose { (res : (List[A],List[A])) ⇒ splitSpec(list, res) } }
```

// Solution:

```
def split[A](list : List[A]) : (List[A], List[A]) = {
  list match {
    case Nil() ⇒ (List[A](), List[A]())
    case Cons(h, t) ⇒
      val (rec_1, rec_2) = split[A](t)
      (Cons[A](h, rec_2), rec_1) }
} ensuring {
  (res : (List[A], List[A])) ⇒ splitSpec[A](list, res) }
```

A.6 List.listOfSize

```
def listOfSize(s: BigInt): List[BiGInt] = {
  require(s ≥ 0)
  choose((l: List[BiGInt]) ⇒ l.size == s) }
```

// Solution:

```
def listOfSize(s : BigInt): List[BiGInt] = {
  require(s ≥ BiGInt(0))
  if (s == BiGInt(0)) List[BiGInt]()
  else Cons[BiGInt](s, listOfSize(s - BiGInt(1)))
} ensuring {
  (l : List[BiGInt]) ⇒ l.size == s }
```

A.7 SortedList.insert

```
def isSorted(list: List[BiGInt]): Boolean = list match {
```

Appendix A. Synthesis Benchmarks by Specification

```
case Cons(x1, t@Cons(x2, _)) ⇒ x1 ≤ x2 && isSorted(t)
case _ ⇒ true }
```

```
def insert(in: List[BigInt], v: BigInt): List[BigInt] = {
  require(isSorted(in))
  choose { (out : List[BigInt]) ⇒
    (out.content == in.content ++ Set(v)) && isSorted(out) } }
```

// Solution:

```
def insert(in : List[BigInt], v : BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Nil() ⇒ List(v)
    case Cons(h, t) ⇒
      val rec = insert(t, v)
      if (h == v) rec
      else if (h < v) Cons[BigInt](h, rec)
      else Cons[BigInt](v, Cons[BigInt](h, t)) }
  } ensuring {
    (out : List[BigInt]) ⇒
      out.content == in.content ++ Set[BigInt](v) && isSorted(out) }
```

A.8 SortedList.insertAlways

```
def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) ⇒ x1 ≤ x2 && isSorted(t)
  case _ ⇒ true }
```

```
def insertAlways(in: List[BigInt], v: BigInt) = {
  require(isSorted(in))
  choose { (out : List[BigInt]) ⇒
    (out.content == in.content ++ Set(v)) &&
    isSorted(out) && out.size == in.size + 1 } }
```

// Solution:

```
def insertAlways(in : List[BigInt], v : BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Nil() ⇒
      List(v)
    case Cons(h, t) ⇒
      val rec = insertAlways(t, v)
```



```

    if (h == v) Cons[BigInt](v, rec)
    else if (h < v) Cons[BigInt](h, rec)
    else Cons[BigInt](v, Cons[BigInt](h, t)) }
} ensuring {
  (out : List[BigInt]) =>
    out.content == in.content ++ Set[BigInt](v) &&
    isSorted(out) && out.size == in.size + BigInt(1) }

```

A.9 SortedList.delete

```

def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) => x1 ≤ x2 && isSorted(t)
  case _ => true }

def delete(in: List[BigInt], v: BigInt) = {
  require(isSorted(in))
  choose( (res : List[BigInt]) =>
    (res.content == in.content -- Set(v)) && isSorted(res) ) }

```

// Solution:

```

def delete(in : List[BigInt], v : BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Nil() =>
      List[BigInt]()
    case Cons(h, t) =>
      val rec = delete(t, v)
      if (h == v) rec
      else Cons[BigInt](h, rec) }
} ensuring {
  (res : List[BigInt]) => res.content == in.content -- Set[BigInt](v) && isSorted(res) }

```

A.10 SortedList.union

```

def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) => x1 ≤ x2 && isSorted(t)
  case _ => true }

def insert(in: List[BigInt], v: BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Cons(h, t) =>
      if (v < h) Cons(v, in)

```

```

    else if (v == h) in
    else Cons(h, insert(t, v))
  case Nil() ⇒ Cons(v, Nil[BigInt]()) }
} ensuring { res ⇒
  (res.content == in.content ++ Set(v)) && isSorted(res) }

def union(in1: List[BigInt], in2: List[BigInt]) = {
  require(isSorted(in1) && isSorted(in2))
  choose { (out : List[BigInt]) ⇒
    (out.content == in1.content ++ in2.content) && isSorted(out) } }

```

// Solution:

```

def union(in1 : List[BigInt], in2 : List[BigInt]): List[BigInt] = {
  require(isSorted(in1) && isSorted(in2))
  in1 match {
    case Nil() ⇒ in2
    case Cons(h, t) ⇒ insert(union(t, in2), h) }
} ensuring {
  (out : List[BigInt]) ⇒
    out.content == in1.content ++ in2.content && isSorted(out) }

```

A.11 SortedList.diff

```

def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) ⇒ x1 ≤ x2 && isSorted(t)
  case _ ⇒ true }

def delete(in1: List[BigInt], v: BigInt): List[BigInt] = {
  require(isSorted(in1))
  in1 match {
    case Cons(h,t) ⇒
      if (h < v) Cons(h, delete(t, v))
      else if (h == v) delete(t, v)
      else in1
    case Nil() ⇒ Nil[BigInt]() }
} ensuring { res ⇒
  res.content == in1.content -- Set(v) && isSorted(res) }

def diff(in1: List[BigInt], in2: List[BigInt]) = {
  require(isSorted(in1) && isSorted(in2))
  choose { (out : List[BigInt]) ⇒
    (out.content == in1.content -- in2.content) && isSorted(out) } }

```

// Solution:

```
def diff(in1 : List[BigInt], in2 : List[BigInt]): List[BigInt] = {
  require(isSorted(in1) && isSorted(in2))
  in2 match {
    case Nil() => in1
    case Cons(h, t) => delete(diff(in1, t), h) }
} ensuring {
  (out : List[BigInt]) => out.content == in1.content -- in2.content && isSorted(out) }
```

A.12 SortedList.insertionSort

```
def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) => x1 ≤ x2 && isSorted(t)
  case _ => true }
```

```
def insert(in: List[BigInt], v: BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Cons(h, t) =>
      if (v < h) Cons(v, in)
      else if (v == h) in
      else Cons(h, insert(t, v))
    case Nil() => Cons(v, Nil()) }
} ensuring { res =>
  (res.content == in.content ++ Set(v)) && isSorted(res) }
```

```
def insertionSort(in: List[BigInt]): List[BigInt] = {
  choose { (out: List[BigInt]) =>
    out.content == in.content && isSorted(out) } }
```

// Solution:

```
def insertionSort(in : List[BigInt]): List[BigInt] = {
  in match {
    case Nil() => List[BigInt]()
    case Cons(h, t) => insert(insertionSort(t), h) }
} ensuring {
  (out : List[BigInt]) => out.content == in.content && isSorted(out) }
```

A.13 StrictSortedList.insert

```
def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) => x1 < x2 && isSorted(t)
```

```
case _ ⇒ true }
```

```
def insert(in: List[BigInt], v: BigInt): List[BigInt] = {
  require(isSorted(in))
  ???[List[BigInt]]
} ensuring { (out : List[BigInt]) ⇒
  (out.content == in.content ++ Set(v)) && isSorted(out) }
```

// Solution:

```
def insert(in : List[BigInt], v : BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Nil() ⇒ List(v)
    case Cons(h, t) ⇒
      val rec = insert(t, v)
      if (h == v) rec
      else if (h < v) Cons[BigInt](h, rec)
      else Cons[BigInt](v, Cons[BigInt](h, t)) }
} ensuring {
  (out : List[BigInt]) ⇒
    out.content == in.content ++ Set[BigInt](v) && isSorted(out) }
```

A.14 StrictSortedList.delete

```
def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) ⇒ x1 < x2 && isSorted(t)
  case _ ⇒ true }
```

```
def delete(in: List[BigInt], v: BigInt) = {
  require(isSorted(in))
  choose( (res : List[BigInt]) ⇒
    (res.content == in.content -- Set(v)) && isSorted(res) ) }
```

// Solution:

```
def delete(in : List[BigInt], v : BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Nil() ⇒ List[BigInt]()
    case Cons(h, t) ⇒
      val rec = delete(t, v)
      if (h == v) rec
      else Cons[BigInt](h, rec) }
```

```

} ensuring {
  (res : List[BigInt]) =>
    res.content == in.content -- Set[BigInt](v) && isSorted(res) }

```

A.15 StrictSortedList.union

```

def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) => x1 < x2 && isSorted(t)
  case _ => true }

def insert(in1: List[BigInt], v: BigInt): List[BigInt] = {
  require(isSorted(in1))
  in1 match {
    case Cons(h, t) =>
      if (v < h) Cons(v, in1)
      else if (v == h) in1
      else Cons(h, insert(t, v))
    case Nil() => Cons(v, Nil()) }
} ensuring { res =>
  (res.content == in1.content ++ Set(v)) && isSorted(res) }

def union(in1: List[BigInt], in2: List[BigInt]) = {
  require(isSorted(in1) && isSorted(in2))
  choose { (out : List[BigInt]) =>
    (out.content == in1.content ++ in2.content) && isSorted(out) } }

// Solution:
def union(in1 : List[BigInt], in2 : List[BigInt]): List[BigInt] = {
  require(isSorted(in1) && isSorted(in2))
  in1 match {
    case Nil() => in2
    case Cons(h, t) => insert(union(t, in2), h) }
} ensuring {
  (out : List[BigInt]) =>
    out.content == in1.content ++ in2.content && isSorted(out) }

```

A.16 UnaryNumerals.add

```

sealed abstract class Num
case object Z extends Num
case class S(pred: Num) extends Num

def value(n: Num): BigInt = {

```

```

n match {
  case Z ⇒ BigInt(0)
  case S(p) ⇒ BigInt(1) + value(p) }
} ensuring (_ ≥ 0)

```

```

def add(x: Num, y: Num): Num = {
  choose { (r : Num) ⇒
    value(r) == value(x) + value(y) } }

```

// Solution:

```

def add(x : Num, y : Num): Num = {
  x match {
    case Z ⇒ y
    case S(pred) ⇒ S(add(pred, y)) }
} ensuring {
  (r : Num) ⇒ value(r) == value(x) + value(y) }

```

A.17 UnaryNumerals.distinct

```

sealed abstract class Num
case object Z extends Num
case class S(pred: Num) extends Num

```

```

def value(n: Num): BigInt = {
  n match {
    case Z ⇒ BigInt(0)
    case S(p) ⇒ BigInt(1) + value(p) }
} ensuring (_ ≥ 0)

```

```

def add(x: Num, y: Num): Num = {
  x match {
    case S(p) ⇒ S(add(p, y))
    case Z ⇒ y }
} ensuring { (r : Num) ⇒
  value(r) == value(x) + value(y) }

```

```

def distinct(x: Num, y: Num): Num = {
  choose { (r : Num) ⇒
    r != x && r != y } }

```

// Solution:

```

def distinct(x : Num, y : Num): Num = add(y, S(x)) ensuring {

```

```
(r : Num) ⇒ r != x && r != y }
```

A.18 UnaryNumerals.mult

```
sealed abstract class Num
case object Z extends Num
case class S(pred: Num) extends Num
```

```
def value(n: Num): BigInt = {
  n match {
    case Z ⇒ BigInt(0)
    case S(p) ⇒ BigInt(1) + value(p) }
} ensuring (_ ≥ 0)
```

```
def add(x: Num, y: Num): Num = {
  x match {
    case S(p) ⇒ S(add(p, y))
    case Z ⇒ y }
} ensuring { (r : Num) ⇒
  value(r) == value(x) + value(y) }
```

```
def mult(x: Num, y: Num): Num = {
  choose { (r : Num) ⇒
    value(r) == value(x) * value(y) } }
```

// Solution:

```
def mult(x : Num, y : Num): Num = {
  x match {
    case Z ⇒ Z
    case S(pred) ⇒ add(mult(pred, y), y) }
} ensuring {
  (r : Num) ⇒ value(r) == value(x) * value(y) }
```

A.19 BatchedQueue.enqueue

```
case class Queue[T](f: List[T], r: List[T]) {
  def content: Set[T] = f.content ++ r.content
  def size: BigInt = f.size + r.size

  def invariant: Boolean = {
    (f == Nil[T]()) ⇒ (r == Nil[T]()) }

  def toList: List[T] = f ++ r.reverse
```

```

def enqueue(v: T): Queue[T] = {
  require(invariant)
  ???[Queue[T]]
} ensuring { (res: Queue[T]) =>
  res.invariant &&
  res.toList.last == v &&
  res.size == size + 1 &&
  res.content == this.content ++ Set(v) }
}

```

// Solution:

```

def enqueue[T](thiss : Queue[T], v : T): Queue[T] = {
  require(thiss.invariant)
  val Queue(f, r) = thiss
  f match {
    case Nil() =>
      Queue[T](Cons[T](v, Nil[T]()), Nil[T]())
    case Cons(h, t) =>
      Queue[T](Queue[T](Cons[T](h, t), r).toList, Cons[T](v, Nil[T]())) }
} ensuring {
  (res : Queue[T]) =>
  res.invariant &&
  res.toList.last == v &&
  res.size == thiss.size + BigInt(1) &&
  res.content == thiss.content ++ Set[T](v) }

```

A.20 BatchedQueue.dequeue

```

case class Queue[T](f: List[T], r: List[T]) {
  def content: Set[T] = f.content ++ r.content
  def size: BigInt = f.size + r.size
  def isEmpty: Boolean = f.isEmpty && r.isEmpty
  def invariant: Boolean = (f.isEmpty) ==> (r.isEmpty)
  def toList: List[T] = f ++ r.reverse

  def dequeue: Queue[T] = {
    require(invariant && !isEmpty)
    ???[Queue[T]]
  } ensuring { (res: Queue[T]) =>
    res.size == size-1 && res.toList == this.toList.tail && res.invariant }
}

```



```
object Queue {
  // Make this available for synthesis
  def reverse[T](l: List[T]) = l.reverse
}

// Solution: (Note: Leon output was manually simplified)
def dequeue[T](thiss : Queue[T]): Queue[T] = {
  require(thiss.invariant && !thiss.isEmpty)
  val Queue(f @ Cons(h, t), r) = thiss
  t match {
    case Nil() =>
      Queue[T](r.reverse, Nil[T]())
    case Cons(h1, t1) =>
      Queue[T](Cons[T](h1, t1), r) }
} ensuring {
  (res : Queue[T]) =>
    res.size == thiss.size - BigInt(1) &&
    res.toList == thiss.toList.tail &&
    res.invariant }
```

A.21 AddressBook.makeAddressBook

```
case class Address[A](info: A, priv: Boolean)

def allPersonal[A](l: List[Address[A]]): Boolean = l match {
  case Nil() => true
  case Cons(a, l1) =>
    if (a.priv) allPersonal(l1)
    else false }

def allBusiness[A](l: List[Address[A]]): Boolean = l match {
  case Nil() => true
  case Cons(a, l1) =>
    if (a.priv) false
    else allBusiness(l1) }

case class AddressBook[A](business: List[Address[A]], personal: List[Address[A]]) {
  def size: BigInt = business.size + personal.size

  def content: Set[Address[A]] = business.content ++ personal.content
```

Appendix A. Synthesis Benchmarks by Specification

```
def invariant = {
  allPersonal(personal) && allBusiness(business) }
}

def makeAddressBook[A](as: List[Address[A]]): AddressBook[A] = {
  choose( (res: AddressBook[A]) => res.content == as.content && res.invariant ) }

// Solution:
def makeAddressBook[A](as : AddressList[A]): AddressBook[A] = {
  as match {
    case Nil() =>
      AddressBook[A](Nil[A](), Nil[A]())
    case Cons(a @ Address(info, priv), tail) =>
      val AddressBook(business, personal) = makeAddressBook[A](tail)
      if (priv) {
        AddressBook[A](business, Cons[A](Address[A](info, true), personal))
      } else {
        AddressBook[A](Cons[A](Address[A](info, false), business), personal) } }
  } ensuring {
    (res : AddressBook[A]) => res.content == as.content && res.invariant }
}
```

A.22 AddressBook.merge

```
def union[A](l1: List[A], l2: List[A]): List[A] = { l1 match {
  case Nil() => l2
  case Cons(h, t) => Cons(h, union(t, l2))
}} ensuring { res => res.content == l1.content ++ l2.content }
```

```
case class Address[A](info: A, priv: Boolean)
```

```
def allPersonal[A](l: List[Address[A]]): Boolean = l match {
  case Nil() => true
  case Cons(a, l1) =>
    if (a.priv) allPersonal(l1)
    else false }
}
```

```
def allBusiness[A](l: List[Address[A]]): Boolean = l match {
  case Nil() => true
  case Cons(a, l1) =>
    if (a.priv) false
    else allBusiness(l1) }
}
```

```

case class AddressBook[A](business: List[Address[A]], personal: List[Address[A]]) {
  def size: BigInt = business.size + personal.size
  def content: Set[Address[A]] = business.content ++ personal.content
  def invariant = allPersonal(personal) && allBusiness(business)
}

```

```

def merge[A](a1: AddressBook[A], a2: AddressBook[A]): AddressBook[A] = {
  require(a1.invariant && a2.invariant)
  ???[AddressBook[A]]
} ensuring {
  (res: AddressBook[A]) =>
    res.personal.content == (a1.personal.content ++ a2.personal.content) &&
    res.business.content == (a1.business.content ++ a2.business.content) &&
    res.invariant }

```

// Solution:

```

def merge[A](a1 : AddressBook[A], a2 : AddressBook[A]): AddressBook[A] = {
  require(
    (allPersonal[A](a1.personal) && allBusiness[A](a1.business)) &&
    (allPersonal[A](a2.personal) && allBusiness[A](a2.business)) )
  val AddressBook(business, personal) = a2
  val AddressBook(business1, personal1) = a1
  AddressBook[A](business ++ business1, personal1 ++ personal)
} ensuring {
  (res : AddressBook[A]) =>
    res.personal.content == a1.personal.content ++ a2.personal.content &&
    res.business.content == a1.business.content ++ a2.business.content &&
    (allPersonal[A](res.personal) && allBusiness[A](res.business)) }

```

A.23 RunLength.encode

```

def decode[A](l: List[(BigInt, A)]): List[A] = {
  def fill[A](i: BigInt, a: A): List[A] = {
    if (i > 0) a :: fill(i - 1, a)
    else Nil[A]() }

  l match {
    case Nil() => Nil[A]()
    case Cons((i, x), xs) =>
      fill(i, x) ++ decode(xs) }
}

```

Appendix A. Synthesis Benchmarks by Specification

```
def legal[A](l: List[(BigInt, A)]): Boolean = l match {
  case Nil() ⇒ true
  case Cons((i, _), Nil()) ⇒ i > 0
  case Cons((i, x), tl@Cons((_, y), _)) ⇒
    i > 0 && x != y && legal(tl) }

def encode[A](l: List[A]): List[(BigInt, A)] = ???[List[(BigInt, A)]] ensuring {
  (res: List[(BigInt, A)]) ⇒
    legal(res) && decode(res) == l }

// Solution:
def encode[A](l : List[A]): List[(BigInt, A)] = {
  l match {
    case Nil() ⇒ List[(BigInt, A)]()
    case Cons(h, t) ⇒
      encode[A](t) match {
        case Nil() ⇒
          List((BigInt(1), h))
        case Cons(h1 @ (h_1, h_2), t1) ⇒
          if (h == h_2) {
            Cons[(BigInt, A)]((h_1 + BigInt(1), h), t1)
          } else {
            Cons[(BigInt, A)]((BigInt(1), h), Cons[(BigInt, A)](h1, t1)) } } }
  } ensuring {
    (res : List[(BigInt, A)]) ⇒ legal[A](res) && decode[A](res) == l }
}
```

A.24 Diffs.diffs

```
def diffs(l: List[BigInt]): List[BigInt] =
  choose((res: List[BigInt]) ⇒
    res.size == l.size && undiff(res) == l )

def undiff(l: List[BigInt]) = {
  l.scanLeft(BigInt(0))(_ + _).tail }
```

```
// Solution:
def diffs(l : List[BigInt]): List[BigInt] = {
  l match {
    case Nil() ⇒
      List[BigInt]()
    case Cons(h, t) ⇒
      diffs(t) match {
```

```
    case Nil() =>
      Cons[BigInt](h, t)
    case Cons(h1, t1) =>
      Cons[BigInt](h, Cons[BigInt](h1 - h, t1)) } }
} ensuring {
  (res : List[BigInt]) => res.size == l.size && undiff(res) == l }
```


B Synthesis Benchmarks by Example

We present the synthesis by example benchmarks used in Chapter 3. For each benchmark, we give the code containing the synthesis problem (the **choose** construct, or the synthesis hole `???`), followed by the synthesized solution.

B.1 UnaryNumerals.add

```
sealed abstract class Num
case object Z extends Num
case class S(pred: Num) extends Num

def add(x: Num, y: Num): Num = {
  ???[Num]
} ensuring { res =>
  ((x, y), res) passes {
    case (Z, _) => y
    case (_, Z) => x
    case (S(Z), S(Z)) => S(S(Z))
    case (S(S(Z)), S(Z)) => S(S(S(Z))) } }
```

// Solution:

```
def add(x : Num, y : Num): Num = x match {
  case Z => y
  case S(pred) => S(add(pred, y)) }
```

B.2 List.append

```
def append[A](l1: List[A], l2: List[A]) : List[A] = {
  ???[List[A]]
} ensuring { (res: List[A]) =>
```

```
((l1, l2), res) passes {
  case (Nil(), l) ⇒ l
  case (l, Nil()) ⇒ l
  case (Cons(a, Nil()), Cons(b, Nil())) ⇒ Cons(a, Cons(b, Nil()))
  case (Cons(a, Cons(b, Nil())), Cons(c, Nil())) ⇒ Cons(a, Cons(b, Cons(c, Nil())))
  case (Cons(a, Cons(b, Nil())), Cons(c, Cons(d, Nil()))) ⇒
    Cons(a, Cons(b, Cons(c, Cons(d, Nil())))) } }
```

// Solution:

```
def append[A](l1 : List[A], l2 : List[A]): List[A] = l1 match {
  case Nil() ⇒ l2
  case Cons(h, t) ⇒ Cons[A](h, append[A](t, l2)) }
```

B.3 Calc.eval

```
abstract class Expr
case class Const(i: BigInt) extends Expr
case class Plus(l: Expr, r: Expr) extends Expr
case class Minus(l: Expr, r: Expr) extends Expr
case class Times(l: Expr, r: Expr) extends Expr
case class Max(l: Expr, r: Expr) extends Expr

def eval(e: Expr): BigInt = ???[BigInt] ensuring { res ⇒
  (e, res) passes {
    case Const(i) ⇒ i
    case Plus(Const(BigInt(8)), Const(BigInt(5))) ⇒ 13
    case Plus(Const(BigInt(-10)), Const(BigInt(7))) ⇒ -3
    case Minus(Const(BigInt(-10)), Const(BigInt(7))) ⇒ -17
    case Minus(Const(BigInt(8)), Const(BigInt(5))) ⇒ 3
    case Times(Const(BigInt(8)), Const(BigInt(5))) ⇒ 40
    case Times(Const(BigInt(-10)), Const(BigInt(7))) ⇒ -70
    case Max(Const(BigInt(18)), Const(BigInt(5))) ⇒ 18
    case Max(Const(BigInt(8)), Const(BigInt(8))) ⇒ 8
    case Max(Const(BigInt(-5)), Const(BigInt(5))) ⇒ 5
    case Max(Const(BigInt(-8)), Const(BigInt(22))) ⇒ 22 } }
```

// Solution:

```
def eval(e : Expr): BigInt = e match {
  case Const(i) ⇒ i
  case Max(l, r) ⇒
    val rec = eval(l)
    val rec1 = eval(r)
```



```

    if (rec == rec1) rec1
    else if (rec < rec1) rec1
    else rec
  case Minus(l, r) =>
    eval(l) - eval(r)
  case Plus(l, r) =>
    eval(l) + eval(r)
  case Times(l, r) =>
    eval(l) * eval(r)
}

```

B.4 Tree.countLeaves

```

abstract class Tree[A]
case class Leaf[A]() extends Tree[A]
case class Node[A](l: Tree[A], v: A, r: Tree[A]) extends Tree[A]

```

```

def countLeaves[A](t: Tree[A]): BigInt = ???[BigInt] ensuring { (res: BigInt) =>
  (t, res) passes {
    case Leaf() => 1
    case Node(Leaf(), _, Leaf()) => 2
    case Node(Node(Leaf(), _, Leaf()), _, Leaf()) => 3
    case Node(Node(Leaf(), _, Leaf()), _, Node(Leaf(), _, Leaf())) => 4 } }

```

```

def countLeaves[A](t: Tree[A]): BigInt = t match {
  case Leaf() =>
    BigInt(1)
  case Node(l, v, r) =>
    countLeaves[A](l) + countLeaves[A](r) }

```

B.5 Dictionary.replace

```

def dictReplace[A, B](l: List[(A, B)], key: A, v: B): List[(A, B)] = {
  ???[List[(A, B)]]
} ensuring { res =>
  ((l, key, v), res) passes {
    case (Nil(), _, _) => Nil()
    case (Cons((k1, v1), Nil()), k, v) if k == k1 => List((k1, v))
    case (Cons((k1, v1), Nil()), k, v) if k != k1 => l
    case (Cons((k1, v1), Cons((k2, v2), Nil())), k, v) if k != k1 && k == k2 =>
      List((k1, v1), (k2, v))
    case (Cons((k1, v1), Cons((k2, v2), Nil())), k, v) if k != k1 && k != k2 =>
      List((k1, v1), (k2, v2))
  }

```

```
case (Cons((k1, v1), Cons((k2, v2), Nil())), k, v) if k == k1 && k == k2 =>
  List((k1, v), (k2, v)) } }
```

// Solution:

```
def dictReplace[A, B](l : List[(A, B)], key : A, v : B): List[(A, B)] = l match {
  case Nil() => List[(A, B)]()
  case Cons(h @ (h_1, h_2), t) =>
    val rec = dictReplace[A, B](t, key, v)
    if (h_1 == key) Cons[(A, B)]((key, v), rec)
    else Cons[(A, B)](h, rec) }
```

B.6 Dictionary.find

```
def dictFind[A, B](l : List[(A, B)], key : A): Option[B] = ???[Option[B]] ensuring { res =>
  ((l, key), res) passes {
    case (Nil(), _) => None()
    case (Cons((k1, v1), Nil()), k) if k == k1 => Some(v1)
    case (Cons((k1, v1), Nil()), k) if k != k1 => None()
    case (Cons((k1, v1), Cons((k2, v2), Nil())), k) if k == k1 => Some(v1)
    case (Cons((k1, v1), Cons((k2, v2), Nil())), k) if k != k1 && k == k2 => Some(v2)
    case (Cons((k1, v1), Cons((k2, v2), Nil())), k) if k != k1 && k != k2 => None() } }
```

// Solution:

```
def dictFind[A, B](l : List[(A, B)], key : A): Option[B] = l match {
  case Nil() => None[B]()
  case Cons(h @ (h_1, h_2), t) =>
    if (h_1 == key) Some[B](h_2)
    else dictFind[A, B](t, key) }
```

B.7 List.diffs

```
def diffs(l : List[BigInt]): List[BigInt] = {
  ???[List[BigInt]]
} ensuring { (res : List[BigInt]) =>
  (l, res) passes {
    case Nil() => Nil()
    case Cons(BigInt(55), Nil()) => List(55)
    case Cons(BigInt(100), Cons(BigInt(-100), Nil())) => List(200, -100)
    case Cons(BigInt(1), Cons(BigInt(2), Cons(BigInt(22), Nil()))) => List(-1, -20, 22) } }
```

// Solution:

```
def diffs(l : List[BigInt]): List[BigInt] = l match {
  case Nil() => List[BigInt]()
```

```

case Cons(h, t) ⇒ t match {
  case Nil() ⇒ List(h)
  case Cons(h1, t1) ⇒
    Cons[BigInt](h - h1, diffs(t)) } }

```

B.8 Expr.fv

```

abstract class Expr
case class Var(i: BigInt) extends Expr
case class Unit() extends Expr
case class App(f: Expr, a: Expr) extends Expr
case class Lam(v: BigInt, b: Expr) extends Expr
case class Let(v: BigInt, vl: Expr, b: Expr) extends Expr

def fv(e: Expr): Set[BigInt] = ???[Set[BigInt]] ensuring { res ⇒
  (e, res) passes {
    case Var(i) ⇒ Set(i)
    case App(Var(i), Var(j)) ⇒ Set(i, j)
    case Lam(BigInt(0), Var(BigInt(0))) ⇒ Set()
    case Lam(BigInt(0), Var(BigInt(1))) ⇒ Set(BigInt(1))
    case Let(BigInt(0), Var(BigInt(0)), Var(BigInt(1))) ⇒ Set(BigInt(0), BigInt(1))
    case Let(BigInt(0), Var(BigInt(1)), Var(BigInt(0))) ⇒ Set(BigInt(1))
    case Let(BigInt(0), Var(BigInt(1)), Var(BigInt(2))) ⇒ Set(BigInt(1), BigInt(2)) } }

```

// Solution:

```

def fv(e : Expr): Set[BigInt] = e match {
  case Unit() ⇒
    Set[BigInt]()
  case Var(i) ⇒
    Set[BigInt](i)
  case Lam(v, b) ⇒
    fv(b) -- Set[BigInt](v)
  case App(f, a) ⇒
    fv(a) ++ fv(f)
  case Let(v, vl, b) ⇒
    (fv(b) -- Set[BigInt](v)) ++ fv(vl)
}

```

B.9 UnaryNumerals.isEven

```

sealed abstract class Num
case object Z extends Num
case class S(pred: Num) extends Num

```

```
def isEven(x: Num): Boolean = ???[Boolean] ensuring { res =>
  (x, res) passes {
    case Z => true
    case S(Z) => false
    case S(S(Z)) => true } }
```

// Solution:

```
def isEven(x : Num): Boolean = x match {
  case Z => true
  case S(pred) => !isEven(pred) }
```

B.10 SortedList.insert

```
def isSorted(list: List[BigInt]): Boolean = list match {
  case Cons(x1, t@Cons(x2, _)) => x1 < x2 && isSorted(t)
  case _ => true }
```

```
def insert(in: List[BigInt], v: BigInt): List[BigInt] = {
  require(isSorted(in))
  choose { (out : List[BigInt]) =>
    ((in, v), out) passes {
      case (Nil(), v) => List(v)
      case (Cons(a, Nil()), b) if a == b => in
      case (Cons(BigInt(1), Nil()), BigInt(2)) => List(1, 2)
      case (Cons(BigInt(1), Nil()), BigInt(-22)) => List(-22, 1)
      case (Cons(BigInt(42), Nil()), BigInt(-22)) => List(-22, 42)
      case (Cons(BigInt(1), Cons(BigInt(42), Nil())), BigInt(100)) =>
        List(1, 42, 100) } } }
```

// Solution:

```
def insert(in : List[BigInt], v : BigInt): List[BigInt] = {
  require(isSorted(in))
  in match {
    case Nil() =>
      List(v)
    case Cons(h, t) =>
      val rec = insert(t, v)
      if (h == v) rec
      else if (h < v) Cons[BigInt](h, rec)
      else List(v, h) } }
```

B.11 UnaryNumerals.mult

```
sealed abstract class Num
case object Z extends Num
case class S(pred: Num) extends Num

def add(x: Num, y: Num): Num = x match {
  case S(p) ⇒ S(add(p, y))
  case Z ⇒ y }

def mult(x: Num, y: Num): Num = ???[Num] ensuring { res ⇒
  ((x, y), res) passes {
    case (Z, _) ⇒ Z
    case (_, Z) ⇒ Z
    case (S(Z), S(Z)) ⇒ S(Z)
    case (S(S(S(Z))), S(S(Z))) ⇒ (S(S(S(S(S(S(Z))))))) } }

// Solution:
def mult(x: Num, y: Num): Num = x match {
  case Z ⇒ Z
  case S(pred) ⇒ add(mult(pred, y), y) }
```

B.12 Tree.postorder

```
abstract class Tree[A]
case class Leaf[A]() extends Tree[A]
case class Node[A](l: Tree[A], v: A, r: Tree[A]) extends Tree[A]

def append[A](l1: List[A], l2: List[A]) = l1 ++ l2

def postorder[A](t: Tree[A]): List[A] = ???[List[A]] ensuring { res ⇒
  (t, res) passes {
    case Leaf() ⇒ Nil()
    case Node(Leaf(), a, Leaf()) ⇒ Cons(a, Nil())
    case Node(Node(Leaf(), a, Leaf()), b, Leaf()) ⇒ Cons(a, Cons(b, Nil()))
    case Node(Leaf(), a, Node(Leaf(), b, Leaf())) ⇒ Cons(a, Cons(b, Nil())) } }

// Solution:
def postorder[A](t: Tree[A]): List[A] = t match {
  case Leaf() ⇒ List[A]()
  case Node(l, v, r) ⇒ append[A](postorder[A](l), Cons[A](v, postorder[A](r))) }
```

B.13 List.reserve

```
def append[A](l1: List[A], l2: List[A]) = l1 ++ l2

def reserve[A](l: List[A]): List[A] = ???[List[A]] ensuring { res =>
  (l, res) passes {
    case Nil() => Nil()
    case Cons(a, Nil()) => Cons(a, Nil())
    case Cons(a, Cons(b, Nil())) => Cons(b, Cons(a, Nil()))
    case Cons(a, Cons(b, Cons(c, Nil()))) => Cons(c, Cons(b, Cons(a, Nil()))) } }

// Solution:
def reserve[A](l: List[A]): List[A] = l match {
  case Nil() => List[A]()
  case Cons(h, t) => append[A](reserve[A](t), List(h)) }
```

B.14 RunLength.encode

```
def encode[A](l: List[A]): List[(BigInt, A)] = ???[List[(BigInt, A)]] ensuring {
  (res: List[(BigInt, A)]) => (l, res) passes {
    case Nil() => Nil()
    case Cons(a, Nil()) =>
      List((1,a))
    case Cons(a, Cons(b, Nil())) if a == b =>
      List((2,a))
    case Cons(a, Cons(b, Cons(c, Nil()))) if a == b && a == c =>
      List((3,a))
    case Cons(a, Cons(b, Cons(c, Nil()))) if a == b && a != c =>
      List((2,a), (1,c))
    case Cons(a, Cons(b, Cons(c, Nil()))) if a != b && b == c =>
      List((1,a), (2,b))
    case Cons(a, Cons(b, Nil())) if a != b =>
      List((1,a), (1,b)) } }
```

```
// Solution
def encode[A](l: List[A]): List[(BigInt, A)] = l match {
  case Nil() => List[(BigInt, A)]()
  case Cons(h, t) =>
    encode[A](t) match {
      case Nil() =>
        List((BigInt(1), h))
      case Cons(h1 @ (h_1, h_2), t1) =>
```

```

    if (h == h_2) {
      Cons[(BigInt, A)]((h_1 + BigInt(1), h), t1)
    } else {
      Cons[(BigInt, A)]((BigInt(1), h), Cons[(BigInt, A)](h1, t1)) } } }

```

B.15 List.take

```

def take[A](l: List[A], n: BigInt) : List[A] = {
  require(n ≥ 0)
  ???[List[A]]
} ensuring { (res: List[A]) ⇒
  ((l, n), res) passes {
    case (Nil(), _) ⇒ Nil()
    case (_, BigInt(0)) ⇒ Nil()
    case (Cons(a, Cons(b, Nil())), BigInt(1)) ⇒ Cons(a, Nil())
    case (Cons(a, Cons(b, Nil())), BigInt(2)) ⇒ Cons(a, Cons(b, Nil()))
    case (Cons(a, Cons(b, Nil())), BigInt(5)) ⇒ Cons(a, Cons(b, Nil()))
    case (Cons(a, Cons(b, Cons(c, Nil()))), BigInt(2)) ⇒ Cons(a, Cons(b, Nil())) } }

```

// Solution:

```

def take[A](l : List[A], n : BigInt): List[A] = {
  require(n ≥ BigInt(0))
  if (n == BigInt(0)) List[A]() else {
    l match {
      case Nil() ⇒ List[A]()
      case Cons(h, t) ⇒ Cons[A](h, take[A](t, n - BigInt(1))) } } }

```

B.16 List.unzip

```

def unzip[A, B](l: List[(A, B)]): (List[A], List[B]) = {
  ???[(List[A], List[B])]
} ensuring { res ⇒ (l, res) passes {
  case Nil() ⇒ (Nil(), Nil())
  case Cons((a, b), Nil()) ⇒ (Cons(a, Nil()), Cons(b, Nil()))
  case Cons((a, b), Cons((c, d), Nil())) ⇒ (Cons(a, Cons(c, Nil())), Cons(b, Cons(d, Nil()))) } }

```

// Solution:

```

def unzip[A, B](l : List[(A, B)]): (List[A], List[B]) = l match {
  case Nil() ⇒ (List[A](), List[B]())
  case Cons(h @ (h_1, h_2), t) ⇒
    val (rec_1, rec_2) = unzip[A, B](t)
    (Cons[A](h_1, rec_1), Cons[B](h_2, rec_2)) }

```


C Repair Benchmarks

We present the benchmarks used in Chapter 4. The correct version of each benchmark is listed, along with a set of numbered comments describing the error introduced in each erroneous version of the benchmark. The order is the same as presented in Table 4.1.

C.1 Compiler Benchmark

```
package compiler
```

```
import leon.lang._
import leon.annotation._
import leon.collection._
import leon._
```

```
object Trees {
  abstract class Expr
  case class Plus(lhs: Expr, rhs: Expr) extends Expr
  case class Minus(lhs: Expr, rhs: Expr) extends Expr
  case class LessThan(lhs: Expr, rhs: Expr) extends Expr
  case class And(lhs: Expr, rhs: Expr) extends Expr
  case class Or(lhs: Expr, rhs: Expr) extends Expr
  case class Not(e : Expr) extends Expr
  case class Eq(lhs: Expr, rhs: Expr) extends Expr
  case class Ite(cond: Expr, thn: Expr, els: Expr) extends Expr
  case class IntLiteral(v: BigInt) extends Expr
  case class BoolLiteral(b : Boolean) extends Expr
}
```

```
object Types {
  abstract class Type
```

```
case object IntType extends Type
case object BoolType extends Type
}

object TypeChecker {
  import Trees._
  import Types._

  def typeOf(e : Expr) : Option[Type] = e match {
    case Plus(l,r) => (typeOf(l), typeOf(r)) match {
      case (Some(IntType), Some(IntType)) => Some(IntType)
      case _ => None() }
    case Minus(l,r) => (typeOf(l), typeOf(r)) match {
      case (Some(IntType), Some(IntType)) => Some(IntType)
      case _ => None() }
    case LessThan(l,r) => ( typeOf(l), typeOf(r)) match {
      case (Some(IntType), Some(IntType)) => Some(BoolType)
      case _ => None() }
    case And(l,r) => ( typeOf(l), typeOf(r)) match {
      case (Some(BoolType), Some(BoolType)) => Some(BoolType)
      case _ => None() }
    case Or(l,r) => ( typeOf(l), typeOf(r)) match {
      case (Some(BoolType), Some(BoolType)) => Some(BoolType)
      case _ => None() }
    case Not(e) => typeOf(e) match {
      case Some(BoolType) => Some(BoolType)
      case _ => None() }
    case Eq(lhs, rhs) => (typeOf(lhs), typeOf(rhs)) match {
      case (Some(t1), Some(t2)) if t1 == t2 => Some(BoolType)
      case _ => None() }
    case lte(c, th, el) => (typeOf(c), typeOf(th), typeOf(el)) match {
      case (Some(BoolType), Some(t1), Some(t2)) if t1 == t2 => Some(t1)
      case _ => None() }
    case IntLiteral(_) => Some(IntType)
    case BoolLiteral(_) => Some(BoolType)
  }

  def typeChecks(e : Expr) = typeOf(e).isDefined
}

object Semantics {
  import Trees._
```

```

import Types._
import TypeChecker._

def seml(t : Expr) : BigInt = {
  require( typeOf(t) == ( Some(IntType) : Option[Type] ))
  t match {
    case Plus(lhs , rhs) ⇒ seml(lhs) + seml(rhs)
    case Minus(lhs , rhs) ⇒ seml(lhs) - seml(rhs)
    case lte(cond, thn, els) ⇒
      if (semB(cond)) seml(thn) else seml(els)
    case IntLiteral(v) ⇒ v } }

def semB(t : Expr) : Boolean = {
  require( (Some(BoolType): Option[Type]) == typeOf(t))
  t match {
    case And(lhs, rhs ) ⇒ semB(lhs) && semB(rhs)
    case Or(lhs , rhs ) ⇒ semB(lhs) || semB(rhs)
    case Not(e) ⇒ !semB(e)
    case LessThan(lhs, rhs) ⇒ seml(lhs) < seml(rhs)
    case lte(cond, thn, els) ⇒
      if (semB(cond)) semB(thn) else semB(els)
    case Eq(lhs, rhs) ⇒ (typeOf(lhs), typeOf(rhs)) match {
      case ( Some(IntType), Some(IntType) ) ⇒ seml(lhs) == seml(rhs)
      case ( Some(BoolType), Some(BoolType) ) ⇒ semB(lhs) == semB(rhs)
    }
    case BoolLiteral(b) ⇒ b } }

def b2i(b : Boolean): BigInt = if (b) 1 else 0

@induct
def semUntyped( t : Expr) : BigInt = { t match {
  case Plus (lhs, rhs) ⇒ semUntyped(lhs) + semUntyped(rhs)
  case Minus(lhs, rhs) ⇒ semUntyped(lhs) - semUntyped(rhs)
  case And (lhs, rhs) ⇒ if (semUntyped(lhs)!=0) semUntyped(rhs) else BigInt(0)
  case Or(lhs, rhs ) ⇒
    if (semUntyped(lhs) == 0) semUntyped(rhs) else BigInt(1) // (7) full case
  case Not(e) ⇒
    b2i(semUntyped(e) == 0)
  case LessThan(lhs, rhs) ⇒
    b2i(semUntyped(lhs) < semUntyped(rhs))
  case Eq(lhs, rhs) ⇒
    b2i(semUntyped(lhs) == semUntyped(rhs))

```

```

    case lte(cond, thn, els) =>
      if (semUntyped(cond) == 0) semUntyped(els) else semUntyped(thn)
    case IntLiteral(v) => v
    case BoolLiteral(b) => b2i(b)
  }} ensuring { res => typeOf(t) match {
    case Some(IntType) => res == semI(t)
    case Some(BoolType) => res == b2i(semB(t))
    case None() => true
  }}
}

object Desugar {
  import Types._
  import TypeChecker._
  import Semantics.b2i

  abstract class SimpleE
  case class Plus(lhs : SimpleE, rhs : SimpleE) extends SimpleE
  case class Neg(arg : SimpleE) extends SimpleE
  case class lte(cond : SimpleE, thn : SimpleE, els : SimpleE) extends SimpleE
  case class Eq(lhs : SimpleE, rhs : SimpleE) extends SimpleE
  case class LessThan(lhs : SimpleE, rhs : SimpleE) extends SimpleE
  case class Literal(i : BigInt) extends SimpleE

  @induct
  def desugar(e : Trees.Expr) : SimpleE = { e match {
    case Trees.Plus (lhs, rhs) =>
      Plus(desugar(lhs), desugar(rhs))
      // (1) Full case
    case Trees.Minus(lhs, rhs) =>
      Plus(desugar(lhs), Neg(desugar(rhs))) // (2) Full case
    case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
    case Trees.And (lhs, rhs) => lte(desugar(lhs), desugar(rhs), Literal(0))
    case Trees.Or (lhs, rhs) => lte(desugar(lhs), Literal(1), desugar(rhs))
    case Trees.Not(e) =>
      lte(desugar(e), Literal(0), Literal(1))
      // (4) 1 instead of 0
    case Trees.Eq(lhs, rhs) =>
      Eq(desugar(lhs), desugar(rhs))
    case Trees.lte(cond, thn, els) =>
      lte(desugar(cond), desugar(thn), desugar(els))
  }}

```

```

    // (3) subst. arguments
    // (5) is (3) and (4) combined
    case Trees.IntLiteral(v) ⇒ Literal(v)
    case Trees.BoolLiteral(b) ⇒ Literal(b2i(b))
  }} ensuring { res ⇒
    sem(res) == Semantics.semUntyped(e) }

def sem(e : SimpleE) : BigInt = e match {
  case Plus(lhs, rhs) ⇒ sem(lhs) + sem(rhs)
  case Lte(cond, thn, els) ⇒ if (sem(cond) != 0) sem(thn) else sem(els)
  case Neg(arg) ⇒ -sem(arg)
  case Eq(lhs, rhs) ⇒ b2i(sem(lhs) == sem(rhs))
  case LessThan(lhs, rhs) ⇒ b2i(sem(lhs) < sem(rhs))
  case Literal(i) ⇒ i }
}

object Evaluator {
  import Trees._

  def bToi(b: Boolean): BigInt = if (b) 1 else 0
  def iTob(i: BigInt) = i == 1

  def eval(e: Expr): BigInt = {
    e match {
      case Plus(lhs, rhs) ⇒ eval(lhs) + eval(rhs)
      case Minus(lhs, rhs) ⇒ eval(lhs) - eval(rhs)
      case LessThan(lhs, rhs) ⇒ bToi(eval(lhs) < eval(rhs))
      case And(lhs, rhs) ⇒ bToi(iTob(eval(lhs)) && iTob(eval(rhs)))
      case Or(lhs, rhs) ⇒ bToi(iTob(eval(lhs)) || iTob(eval(rhs)))
      case Not(e) ⇒ bToi(!iTob(eval(e)))
      case Eq(lhs, rhs) ⇒ bToi(eval(lhs) == eval(rhs))
      case Lte(cond, thn, els) ⇒ if (iTob(eval(cond))) eval(thn) else eval(els)
      case IntLiteral(v) ⇒ v
      case BoolLiteral(b) ⇒ bToi(b) } }
  }

  object Simplifier {
    import Trees._
    import Evaluator._

    @induct
    def simplify(e: Expr): Expr = { e match {

```

```

case And(BoolLiteral(false), _) ⇒ BoolLiteral(false)
case Or(BoolLiteral(true), _) ⇒ BoolLiteral(true)
case Plus(IntLiteral(a), IntLiteral(b)) ⇒ IntLiteral(a+b) // (6) a-b
case Not(Not(Not(a))) ⇒ Not(a)
case e ⇒ e
}} ensuring { res ⇒ eval(res) == eval(e) }
}

```

C.2 Heap Benchmark

```

import leon.lang._
import leon.collection._

object Heaps {

  sealed abstract class Heap {
    val rank : BigInt = this match {
      case Leaf() ⇒ 0
      case Node(_, l, r) ⇒
        1 + max(l.rank, r.rank) }

    def content : Set[BigInt] = this match {
      case Leaf() ⇒ Set[BigInt]()
      case Node(v,l,r) ⇒ l.content ++ Set(v) ++ r.content }
  }

  case class Leaf() extends Heap
  case class Node(value: BigInt, left: Heap, right: Heap) extends Heap

  def max(i1: BigInt, i2: BigInt) = if (i1 ≥ i2) i1 else i2

  def hasHeapProperty(h : Heap) : Boolean = h match {
    case Leaf() ⇒ true
    case Node(v, l, r) ⇒
      ( l match {
        case Leaf() ⇒ true
        case n@Node(v2,_,_) ⇒ v ≥ v2 && hasHeapProperty(n)
      }) &&
      ( r match {
        case Leaf() ⇒ true
        case n@Node(v2,_,_) ⇒ v ≥ v2 && hasHeapProperty(n)
      }) }
  }
}

```

```

def hasLeftistProperty(h: Heap) : Boolean = h match {
  case Leaf() ⇒ true
  case Node(_, l, r) ⇒
    hasLeftistProperty(l) &&
    hasLeftistProperty(r) &&
    l.rank ≥ r.rank }

def heapSize(t: Heap): BigInt = { t match {
  case Leaf() ⇒ BigInt(0)
  case Node(v, l, r) ⇒ heapSize(l) + 1 + heapSize(r)
}} ensuring(_ ≥ 0)

private def merge(h1: Heap, h2: Heap) : Heap = {
  require(
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&
    hasHeapProperty(h1) && hasHeapProperty(h2)
  )
  (h1, h2) match {
    case (Leaf(), _) ⇒ h2
    case (_, Leaf()) ⇒ h1 // (2) h2
                          // (6) Swapped first 2 cases
    case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒
      if(v1 ≥ v2) // (1)/(3) swapped the branches/condition
                  // (5) completely wrong condition
        makeN(v1, l1, merge(r1, h2))
      else
        makeN(v2, l2, merge(h1, r2)) // (4) l1 → l2
  }
} ensuring { res ⇒
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(h1) + heapSize(h2) == heapSize(res) &&
  h1.content ++ h2.content == res.content
}

private def makeN(value: BigInt, left: Heap, right: Heap) : Heap = {
  require(hasLeftistProperty(left) && hasLeftistProperty(right))
  if(left.rank ≥ right.rank) // (8) Unnecessary additive constant
    Node(value, left, right)
  else
    Node(value, right, left)
} ensuring { res ⇒ hasLeftistProperty(res) }

```

```

def insert(element: BigInt, heap: Heap) : Heap = {
  require(hasLeftistProperty(heap) && hasHeapProperty(heap))
  merge(Node(element, Leaf(), Leaf()), heap) // (7) element+1
} ensuring { res =>
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(res) == heapSize(heap) + 1 &&
  res.content == heap.content ++ Set(element) }

def findMax(h: Heap) : Option[BiInt] = {
  h match {
    case Node(m,_,_) => Some(m)
    case Leaf() => None() } }

def removeMax(h: Heap) : Heap = {
  require(hasLeftistProperty(h) && hasHeapProperty(h))
  h match {
    case Node(_,l,r) => merge(l, r)
    case l => l }
} ensuring { res =>
  hasLeftistProperty(res) && hasHeapProperty(res) }
}

```

C.3 List Benchmark

```

package leon.custom

import leon._
import leon.lang._
import leon.collection._
import leon.annotation._

sealed abstract class List[T] {
  def size: BigInt = (this match {
    case Nil() => BigInt(0)
    case Cons(h, t) => BigInt(1) + t.size // (9) + 3 instead of +1
  }) ensuring { res => res ≥ 0 && (this, res) passes {
    case Nil() => 0
    case Cons(_, Nil()) => 1
    case Cons(_, Cons(_, Nil())) => 2
  }}

  def content: Set[T] = this match {

```



```

    case Nil() ⇒ Set()
    case Cons(h, t) ⇒ Set(h) ++ t.content }

def contains(v: T): Boolean = (this match {
  case Cons(h, t) if h == v ⇒ true
  case Cons(_, t) ⇒ t.contains(v)
  case Nil() ⇒ false
}) ensuring { res ⇒ res == (content contains v) }

def ++(that: List[T]): List[T] = (this match {
  case Nil() ⇒ that
  case Cons(x, xs) ⇒ Cons(x, xs ++ that) // (2) Forgot x
}) ensuring { res ⇒
  (res.content == this.content ++ that.content) &&
  (res.size == this.size + that.size) }

def head: T = {
  require(this != Nil[T]())
  this match {
    case Cons(h, t) ⇒ h } }

def tail: List[T] = {
  require(this != Nil[T]())
  this match {
    case Cons(h, t) ⇒ t } }

def apply(index: BigInt): T = {
  require(0 ≤ index && index < size)
  if (index == 0) head
  else tail(index-1) }

def ::(t:T): List[T] = Cons(t, this)

def :+(t:T): List[T] = {
  this match {
    case Nil() ⇒ Cons(t, this)
    case Cons(x, xs) ⇒ Cons(x, xs :+ (t)) // (3) Forgot t
  }
} ensuring (res ⇒ (res.size == size + 1) && (res.content == content ++ Set(t)))

def reverse: List[T] = { this match {
  case Nil() ⇒ this

```

Appendix C. Repair Benchmarks

```
    case Cons(x, xs) => xs.reverse :+ x
  }} ensuring (res =>
    (res.size == size) && (res.content == content))

def take(i: BigInt): List[T] = (this, i) match {
  case (Nil(), _) => Nil()
  case (Cons(h, t), i) =>
    if (i == 0) Nil()
    else Cons(h, t.take(i-1)) }

def drop(i: BigInt): List[T] = (this, i) match {
  case (Nil(), _) => Nil()
  case (Cons(h, t), i) =>
    if (i == 0) { // (13) swapped condition
      Cons(h, t)
    } else {
      t.drop(i-1) // (12) forgot -1
    } }

def slice(from: BigInt, to: BigInt): List[T] = {
  require(from < to && to < size && from ≥ 0)
  drop(from).take(to-from)
}

def replace(from: T, to: T): List[T] = this match {
  case Nil() => Nil()
  case Cons(h, t) =>
    val r = t.replace(from, to)
    if (h == from) { // (4) reversed condition
      Cons(to, r)
    } else {
      Cons(h, r) } }

private def chunk0(
  s: BigInt, l: List[T], acc: List[T], res: List[List[T]], s0: BigInt
): List[List[T]] = l match {
  case Nil() =>
    if (acc.size > 0) res :+ acc
    else res
  case Cons(h, t) =>
    if (s0 == 0) {
      chunk0(s, l, Nil(), res :+ acc, s)
    }
```

```

    } else {
      chunk0(s, t, acc :+ h, res, s0-1) } }

def chunks(s: BigInt): List[List[T]] = {
  require(s > 0)
  chunk0(s, this, Nil(), Nil(), s) }

def zip[B](that: List[B]): List[(T, B)] = (this, that) match {
  case (Cons(h1, t1), Cons(h2, t2)) =>
    Cons((h1, h2), t1.zip(t2))
  case (_) => Nil() }

def -(e: T): List[T] = this match {
  case Cons(h, t) =>
    if (e == h) {
      t - e // (11) missing rec. call
    } else {
      Cons(h, t - e) }
  case Nil() => Nil() }

def --(that: List[T]): List[T] = this match {
  case Cons(h, t) =>
    if (that.contains(h)) t -- that
    else Cons(h, t -- that)
  case Nil() => Nil() }

def &(that: List[T]): List[T] = this match {
  case Cons(h, t) =>
    if (that.contains(h)) { // (14) completely wrong condition
      Cons(h, t & that)
    } else {
      t & that }
  case Nil() => Nil() }

def pad(s: BigInt, e: T): List[T] = { (this, s) match {
  case (_, s) if s ≤ 0 =>
    this
  case (Nil(), s) =>
    Cons(e, Nil().pad(s-1, e))
  case (Cons(h, t), s) =>
    Cons(h, t.pad(s, e)) // (1) s-1
}} ensuring { res =>

```

Appendix C. Repair Benchmarks

```
((this,s,e), res) passes {  
  case (Cons(a,Nil()), BigInt(2), x) ⇒ Cons(a, Cons(x, Cons(x, Nil()))) } }
```

```
def find(e: T): Option[BigInt] = this match {  
  case Nil() ⇒ None()  
  case Cons(h, t) ⇒  
    if (h == e) { // (8) reversed condition  
      Some(0)  
    } else {  
      t.find(e) match {  
        case None() ⇒ None()  
        case Some(i) ⇒ Some(i+1) } } } // (6)/(7) Forgot +1/ +2 instead of +1
```

```
def init: List[T] = (this match {  
  case Cons(h, Nil()) ⇒ Nil[T]()  
  case Cons(h, t) ⇒ Cons[T](h, t.init)  
  case Nil() ⇒ Nil[T]()  
}) ensuring ( (r: List[T]) ⇒  
  ((r.size < this.size) || (this.size == 0)) )
```

```
def lastOption: Option[T] = this match {  
  case Cons(h, t) ⇒ t.lastOption.orElse(Some(h))  
  case Nil() ⇒ None() }
```

```
def firstOption: Option[T] = this match {  
  case Cons(h, t) ⇒ Some(h)  
  case Nil() ⇒ None() }
```

```
def unique: List[T] = this match {  
  case Nil() ⇒ Nil()  
  case Cons(h, t) ⇒ Cons(h, t.unique - h) }
```

```
def splitAt(e: T): List[List[T]] = split(Cons(e, Nil()))
```

```
def split(seps: List[T]): List[List[T]] = this match {  
  case Cons(h, t) ⇒  
    if (seps.contains(h)) {  
      Cons(Nil(), t.split(seps))  
    } else {  
      val r = t.split(seps)  
      Cons(Cons(h, r.head), r.tail) }  
  case Nil() ⇒ Cons(Nil(), Nil()) }
```

```

def count(e: T): BigInt = { this match {
  case Cons(h, t) => // (15) completely wrong
    if (h == e) 1 + t.count(e) // (5) Forgot +1
    else t.count(e)
  case Nil() => 0
}} ensuring { res =>
  res + (this - e).size == this.size }

```

```

def evenSplit: (List[T], List[T]) = {
  val c = size/2
  (take(c), drop(c)) }

```

```

def insertAt(pos: BigInt, l: List[T]): List[T] = {
  if(pos < 0) {
    insertAt(size + pos, l)
  } else if(pos == 0) {
    l ++ this
  } else {
    this match {
      case Cons(h, t) =>
        Cons(h, t.insertAt(pos-1, l))
      case Nil() =>
        l } } }

```

```

def replaceAt(pos: BigInt, l: List[T]): List[T] = {
  if(pos < 0) {
    replaceAt(size + pos, l)
  } else if(pos == 0) {
    l ++ this.drop(l.size)
  } else {
    this match {
      case Cons(h, t) =>
        Cons(h, t.replaceAt(pos-1, l))
      case Nil() =>
        l } } }

```

```

def rotate(s: BigInt): List[T] = {
  if (s < 0) {
    rotate(size+s)
  } else {
    val s2 = s % size

```

```
drop(s2) ++ take(s2) } }

def isEmpty = this match {
  case Nil() ⇒ true
  case _ ⇒ false }
}

@ignore
object List {
  def apply[T](elems: T*): List[T] = ???
}

@library
object ListOps {
  def flatten[T](ls: List[List[T]]): List[T] = ls match {
    case Cons(h, t) ⇒ h ++ flatten(t)
    case Nil() ⇒ Nil() }

  def isSorted(ls: List[BigInt]): Boolean = ls match {
    case Nil() ⇒ true
    case Cons(_, Nil()) ⇒ true
    case Cons(h1, Cons(h2, _)) if(h1 > h2) ⇒ false
    case Cons(_, t) ⇒ isSorted(t) }

  def sorted(ls: List[BigInt]): List[BigInt] = ls match {
    case Cons(h, t) ⇒ insSort(sorted(t), h)
    case Nil() ⇒ Nil() }

  def insSort(ls: List[BigInt], v: BigInt): List[BigInt] = ls match {
    case Nil() ⇒ Cons(v, Nil())
    case Cons(h, t) ⇒
      if (v ≤ h) Cons(v, t)
      else Cons(h, insSort(t, v)) }

  def sum(l: List[BigInt]): BigInt = { l match {
    case Nil() ⇒ BigInt(0)
    case Cons(x, xs) ⇒ x + sum(xs) // (10) x → 1
  }} ensuring { (l, _) passes {
    case Cons(a, Nil()) ⇒ a
    case Cons(a, Cons(b, Nil())) ⇒ a + b }}
}
```

```
case class Cons[T](h: T, t: List[T]) extends List[T]
case class Nil[T]() extends List[T]
```

C.4 Numerical Benchmark

```
import leon._
import leon.lang._
import leon.annotation._

object Numerical {
  def power(base: BigInt, p: BigInt): BigInt = {
    require(p ≥ BigInt(0))
    if (p == BigInt(0)) {
      BigInt(1)
    } else if (p % BigInt(2) == BigInt(0)) {
      power(base*base, p/BigInt(2))
    } else {
      base*power(base, p-BigInt(1)) // (1) forgot first 'base'
    }
  }
  } ensuring {
    res ⇒ ((base, p), res) passes {
      case (_, BigInt(0)) ⇒ BigInt(1)
      case (b, BigInt(1)) ⇒ b
      case (BigInt(2), BigInt(7)) ⇒ BigInt(128)
      case (BigInt(2), BigInt(10)) ⇒ BigInt(1024) } }

  def gcd(a: BigInt, b: BigInt): BigInt = {
    require(a > BigInt(0) && b > BigInt(0));
    if (a == b) a
    else if (a > b) gcd(a-b, b)
    else gcd(a, b-a)
  } ensuring { res ⇒
    (a%res == BigInt(0)) && (b%res == BigInt(0)) &&
    (((a,b), res) passes {
      case (BigInt(468), BigInt(24)) ⇒ BigInt(12) } ) }

  def moddiv(a: BigInt, b: BigInt): (BigInt, BigInt) = {
    require(a ≥ BigInt(0) && b > BigInt(0));
    if (b > a) {
      (a, BigInt(0)) // (2) a → 1
    } else {
      val (r1, r2) = moddiv(a-b, b)
    }
  }
```

```

    (r1, r2+1) }
  } ensuring {
    res ⇒ b*res._2 + res._1 == a }
}

```

C.5 MergeSort Benchmark

```

package mergesort
import leon.collection._

object MergeSort {

  def split(l : List[BigInt]) : (List[BigInt],List[BigInt]) = { l match {
    case Cons(a, Cons(b, t)) ⇒
      val (rec1, rec2) = split(t)
      (Cons(a, rec1), Cons(b, rec2)) // (1) forgot 'a'
    case other ⇒ (other, Nil[BigInt]())
  }} ensuring { res ⇒
    val (l1, l2) = res
    l1.size ≥ l2.size &&
    l1.size ≤ l2.size + 1 &&
    l1.size + l2.size == l.size &&
    l1.content ++ l2.content == l.content }

  def isSorted(l : List[BigInt]) : Boolean = l match {
    case Cons(x, t@Cons(y, _)) ⇒ x ≤ y && isSorted(t)
    case _ ⇒ true }

  def merge(l1 : List[BigInt], l2 : List[BigInt]) : List[BigInt] = {
    require(isSorted(l1) && isSorted(l2))
    (l1, l2) match {
      case (Cons(h1, t1), Cons(h2,t2)) ⇒
        if (h1 ≤ h2) // (3) reversed condition
          Cons(h1, merge(t1, l2))
        else
          Cons(h2, merge(l1, t2))
          // (2) h2 → h1
          // (4) missing h2
      case (Nil(), _) ⇒ l2 // (5) l2 → l1
      case (_, Nil()) ⇒ l1 }
  } ensuring { res ⇒
    isSorted(res) &&

```



```
    res.size == l1.size + l2.size &&
    res.content == l1.content ++ l2.content }

def mergeSort(l : List[BigInt]) : List[BigInt] = { l match {
  case Nil() => l
  case Cons(_, Nil()) => l
  case other =>
    val (l1, l2) = split(other)
    merge(mergeSort(l1), mergeSort(l2))
}} ensuring { res =>
  isSorted(res) &&
  res.content == l.content &&
  res.size == l.size }
}
```


Bibliography

- [ABBS15] Miltiadis Allamanis, Earl Barr, Christian Bird, and Charles Sutton. Suggesting Accurate Method and Class Names. In *Joint Meeting on Foundations of Software Engineering (FSE)*, pages 38–49, 2015.
- [ABJ⁺13] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8, 2013.
- [AGK13] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive Program Synthesis. In *International Conference on Computer Aided Verification (CAV)*, pages 934–950, 2013.
- [ARU17] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 319–336, 2017.
- [AS14] Miltiadis Allamanis and Charles Sutton. Mining Idioms from Source Code. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 472–483, 2014.
- [BAGP18] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative Code Modeling with Graphs. *CoRR*, abs/1805.08490, 2018.
- [BGB⁺17] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations (ICLR)*, 2017.
- [BKKS13] Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Workshop on Scala (SCALA)*, pages 1:1–1:10, 2013.
- [BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

Bibliography

- [Bla17] Régis William Blanc. *Verification by Reduction to Functional Programs*. PhD thesis, EPFL, Lausanne, 2017.
- [BRV16] Pavol Bielik, Veselin Raychev, and Martin Vechev. PHOG: Probabilistic Model for Code. In *International Conference on Machine Learning (ICML)*, pages 2933–2942, 2016.
- [BT73] T. L. Booth and R. A. Thomson. Applying Probability Measures to Abstract Languages. *IEEE Transactions on Computers*, C-22(5):442–450, 1973.
- [BTGC16] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing Synthesis with Metasketches. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 775–788, 2016.
- [CdlH00] Francisco Casacuberta and Colin de la Higuera. Computational Complexity of Problems on Probabilistic Grammars and Transducers. In *5th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI)*, pages 15–24, 2000.
- [CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.
- [Cla12] Koen Claessen. Shrinking and Showing Functions: (Functional Pearl). In *Proceedings of the 2012 Haskell Symposium*, pages 73–80, 2012.
- [CTBB11] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic Debugging. In *International Conference on Software Engineering (ICSE)*, pages 121–130, 2011.
- [CU77] J Craig Cleaveland and Robert C Uzgalis. *Grammars for Programming Languages*. Elsevier, 1977.
- [FCD15] John Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 229–239, 2015.
- [FL11] Manuel Fähndrich and Francesco Logozzo. Static Contract Checking with Abstract Interpretation. In *Formal Verification of Object-Oriented Software (FoVeOO)*, pages 10–30, 2011.
- [FMBD18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program Synthesis Using Conflict-driven Learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 420–435, 2018.
- [FOWZ16] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-Directed Synthesis: a Type-Theoretic Interpretation. In *ACM SIGPLAN*

- Symposium on Principles of Programming Languages (POPL)*, pages 802–815, 2016.
- [GBC06] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of Boolean Programs with an Application to C. In *International Conference on Computer Aided Verification (CAV)*, pages 358–371, 2006.
- [GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 62–73, 2011.
- [GK15] Tihomir Gvero and Viktor Kuncak. Synthesizing Java Expressions from Free-form Queries. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 416–432, 2015.
- [GKKP13] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete Completion Using Types and Weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 27–38, 2013.
- [GKP11] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive Synthesis of Code Snippets. In *International Conference on Computer Aided Verification (CAV)*, pages 418–423, 2011.
- [GMK11] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-Based Program Repair Using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 173–188, 2011.
- [GNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54–72, 2012.
- [Gre69] Cordell Green. Application of Theorem Proving to Problem Solving. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 219–239, 1969.
- [Ino] The Inox Solver. <https://github.com/epfl-lara/inox>. Accessed 28.07.2018.
- [IQLS15] Jeevana Priya Inala, Xiaokang Qiu, Ben Lerner, and Armando Solar-Lezama. Type Assisted Synthesis of Recursive Transformers on Algebraic Data Types. *CoRR*, 2015.
- [JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program Repair as a Game. In *International Conference on Computer Aided Verification (CAV)*, pages 226–238, 2005.
- [JM08] Daniel Jurafsky and James Martin. *Speech and Language Processing*. Prentice Hall, 2nd edition, 2008.

Bibliography

- [JSGB12] Barbara Jobstmann, Stefan Staber, Andreas Griesmayer, and Roderick Bloem. Finding and Fixing Faults. *Journal of Computer and System Sciences (JCSS)*, 78(2):441–460, 2012.
- [Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [KKK15] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive Program Repair. In *International Conference on Computer Aided Verification (CAV)*, pages 217–233, 2015.
- [KKK16] Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. An Update on Deductive Synthesis and Repair in the Leon Tool. In *Fifth Workshop on Synthesis, SYNT@CAV 2016*, volume 229 of *EPTCS*, pages 100–111, 2016.
- [KKKS13] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis Modulo Recursive Functions. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 407–426, 2013.
- [KKS12] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as Control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 298–315, 2012.
- [Kne16] Etienne Kneuss. *Deductive Synthesis and Repair*. PhD thesis, EPFL, Lausanne, 2016.
- [Knu68] Donald E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
- [Knu77] Donald E. Knuth. A Generalization of Dijkstra’s Algorithm. *Information Processing Letters*, 6(1):1–5, 1977.
- [KRKK17] Manos Koukoutos, Mukund Raghothaman, Etienne Kneuss, and Viktor Kuncak. On Repair with Probabilistic Attribute Grammars. *CoRR*, abs/1707.04148, 2017.
- [LB12] Francesco Logozzo and Thomas Ball. Modular and Verified Automatic Program Repair. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–146, 2012.
- [Leo] Leon Online Interface. <http://leon.epfl.ch>. Accessed 01.06.2018.
- [LET18] Calvin Loncaric, Michael D Ernst, and Emina Torlak. Generalized Data structure Synthesis. In *International Conference on Software Engineering (ICSE)*, pages 958–968, 2018.

-
- [LR15] Fan Long and Martin Rinard. Staged Program Repair with Condition Synthesis. In *10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.
- [LTE16] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast Synthesis of Fast Collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 355–368, 2016.
- [Mit81] Tom M. Mitchell. Generalization as search. In *Readings in Artificial Intelligence*, pages 517–542. Morgan Kaufmann, 1981.
- [MK14] Ravichandhran Madhavan and Viktor Kuncak. Symbolic resource bound inference for functional programs. In *26th International Conference on Computer Aided Verification (CAV)*, pages 762–778, 2014.
- [MKK17] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based Resource Verification for Higher-order Functions with Memoization. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 330–343, 2017.
- [MW71] Zohar Manna and Richard Waldinger. Toward Automatic Program Synthesis. *Communications of the ACM*, 14(3):151–165, 1971.
- [NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.
- [NS08] Mark-Jan Nederhof and Giorgio Satta. Probabilistic Parsing. In *New Developments in Formal Languages and Applications*, volume 113 of *Studies in Computational Intelligence*, pages 229–258. 2008.
- [OZ15] Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 619–630, 2015.
- [PFNM15] Yu Pei, Carlo A. Furia, Martín Nordio, and Bertrand Meyer. Automated Program Repair in an Integrated Development Environment. In *IEEE/ACM International Conference on Software Engineering, (ICSE)*, volume 2, pages 681–684, 2015.
- [PG15] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 107–126, 2015.
- [PGBG12] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-Directed Completion of Partial Expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 275–286, 2012.

Bibliography

- [PGGP14] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, 2014.
- [PKS16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, pages 522–538, 2016.
- [PWF⁺11] Yu Pei, Yi Wei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Code-based Automated Program Fixing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 392–395, 2011.
- [RBV16] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic Model for Code with Decision Trees. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 731–747, 2016.
- [RDK⁺15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Clark Barrett, and Cesare Tinelli. Counterexample Guided Quantifier Instantiation for Synthesis in SMT. In *International Conference on Computer Aided Verification (CAV)*, pages 198–216, 2015.
- [RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 159–169, 2008.
- [RKK17] Andrew Reynolds, Tim King, and Viktor Kuncak. Solving Quantified Linear Arithmetic by Counterexample-Guided Instantiation. *Formal Methods in System Design*, pages 500–532, 2017.
- [RKT⁺17] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. Refutation-Based Synthesis in SMT. *Formal Methods in System Design*, pages 1–30, 2017.
- [RP15] Alex Reinking and Ruzica Piskac. A Type-Directed Approach to Program Repair. In *International Conference on Computer Aided Verification (CAV)*, pages 511–517, 2015.
- [RVK15] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from “Big Code”. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 111–124, 2015.
- [RVY14] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–428, 2014.

-
- [SDE08] Roopsha Samanta, Jyotirmoy V. Deshmukh, and E. Allen Emerson. Automatic Generation of Local Repairs for Boolean Programs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–10, 2008.
 - [SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability Modulo Recursive Programs. In *Static Analysis Symposium (SAS)*, pages 298–315, 2011.
 - [SLTB⁺06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.
 - [SOE14] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. Cost-Aware Automatic Program Repair. In *Static Analysis Symposium (SAS)*, pages 268–284, 2014.
 - [Sta] The Stainless Verifier and Termination Checker. <https://github.com/epfl-lara/stainless>. Accessed 28.07.2018.
 - [Syn] Synquid RunLength benchmark. <http://comcom.csail.mit.edu/demos/#3-rle>. Accessed 21.04.2018.
 - [URD⁺13] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: Specifying Protocols with Concolic Snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 287–296, 2013.
 - [vEJ13] Christian von Essen and Barbara Jobstmann. Program Repair without Regret. In *International Conference on Computer Aided Verification (CAV)*, pages 896–911, 2013.
 - [VKK15] Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. Counter-example complete verification for higher-order functions. In *ACM SIGPLAN Symposium on Scala*, pages 18–29, 2015.
 - [vWMP⁺77] Adriaan van Wijngaarden, Barry James Mailloux, John EL Peck, Cornelis HA Koster, Charles Hodgson Lindsey, Michel Sintzoff, LGLT Meertens, and RG Fisker, editors. *Revised Report on the Algorithmic Language Algol 68*. ACM Sigplan Notices, 1977.
 - [WDS17] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(OOPSLA):62:1–62:26, 2017.
 - [WDS18] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program Synthesis Using Abstraction Refinement. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL):63:1–63:30, 2018.

Bibliography

- [WPF⁺10] Yi Wei, Yu Pei, Carlo A. Furia, Lucas Serpa Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated Fixing of Programs with Contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, 2010.
- [WSK⁺18] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. Systematic Generation of Non-Equivalent Expressions for Relational Algebra. In *International ABZ Conference ASM, Alloy, B, TLA, VDM, Z (ABZ)*, 2018.

Emmanouil Koukoutos

Curriculum Vitae

Personal Information

Preferred first name: **Manos**

Date of Birth: **03/07/1989**

Citizenship: **Greek**

Education

2013-2018

PhD, Computer Science

EPFL, Lausanne, Switzerland

Lab for Automated Reasoning and Analysis (LARA)

Thesis title: *Scaling Functional Synthesis and Repair*

Advisor: Viktor Kuncak

2007-2013

Diploma, Electrical and Computer Engineering

(equivalent to Master of Engineering)

National Technical University of Athens, Athens, Greece

Major: Computer Science

GPA: 9,16 / 10 (“Excellent”)

Publications

Emmanouil Koukoutos and Nikolaos S. Papaspyrou

“Type Inference for a Higher-Order Extension of Prolog”

in *Proceedings of the 9th Panhellenic Logic Symposium, 2013*

Emmanouil Koukoutos and Viktor Kuncak

“Checking Data Structure Properties Orders of Magnitude Faster”

in *RV 2014*

Etienne Kneuss, Manos Koukoutos and Viktor Kuncak

“Deductive Program Repair”

in *CAV 2015*

Manos Koukoutos, Etienne Kneuss and Viktor Kuncak

“An Update on Deductive Synthesis and Repair in the Leon Tool”

in *SYNT 2016*

Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov and Sarfraz Khurshid

“Systematic Generation of Non-Equivalent Expressions for Relational Algebra”

in *ABZ 2018*

Projects

Leon: a verification, synthesis and repair tool for functional programs.

See <https://github.com/epfl-lara/leon>

and <http://leon.epfl.ch>

Amy: a toy functional language for compiler courses, with WebAssembly backend implementation

See <http://lara.epfl.ch/~koukouto/clp17/amy-spec.pdf>

and http://lara.epfl.ch/w/cc17:amy_reference_compiler

Other projects: see <https://github.com/manoskouk/>

Teaching Experience

Algorithms and Complexity | Fall 2011, NTUA

Functional Programming | Fall 2014, EPFL

Advanced Compiler Construction | Spring 2015, 2016 and 2017, EPFL

Computer Language Processing | Fall 2015, 2016 and 2017, EPFL

Mathematical Competitions

3rd prize in the International Mathematical Student Contest “IMC” | 2010

3rd prize in the 24th National Mathematical Olympiad for School Students “Archimedes” | 2007

2nd prize in the Panhellenic Probability Contest for School Students | 2007

2nd prize in the Panhellenic Mathematical Contest for School Students “Euclid” | 2003, 2005 and 2007

